



ANTI-TAMPER METHOD FOR FIELD PROGRAMMABLE GATE ARRAYS  
THROUGH  
DYNAMIC RECONFIGURATION AND DECOY CIRCUITS

THESIS

Samuel J. Stone, Captain, USAF

AFIT/GE/ENG/08-30

DEPARTMENT OF THE AIR FORCE  
AIR UNIVERSITY

**AIR FORCE INSTITUTE OF TECHNOLOGY**

Wright-Patterson Air Force Base, Ohio

APPROVED FOR PUBLIC RELEASE; DISTRIBUTION UNLIMITED.

The views expressed in this thesis are those of the author and do not reflect the official policy or position of the United States Air Force, Department of Defense, or the United States Government.

ANTI-TAMPER METHOD FOR FIELD PROGRAMMABLE GATE  
ARRAYS  
THROUGH  
DYNAMIC RECONFIGURATION AND DECOY CIRCUITS

THESIS

Presented to the Faculty  
Department of Electrical and Computer Engineering  
Graduate School of Engineering and Management  
Air Force Institute of Technology  
Air University  
Air Education and Training Command  
In Partial Fulfillment of the Requirements for the  
Degree of Master of Science in Electrical Engineering

Samuel J. Stone, B.S.C.E.  
Captain, USAF

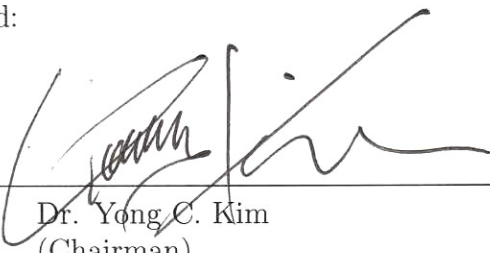
March 2008

APPROVED FOR PUBLIC RELEASE; DISTRIBUTION UNLIMITED.


ANTI-TAMPER METHOD FOR FIELD PROGRAMMABLE GATE  
ARRAYS  
THROUGH  
DYNAMIC RECONFIGURATION AND DECOY CIRCUITS

Samuel J. Stone, B.S.C.E.  
Captain, USAF

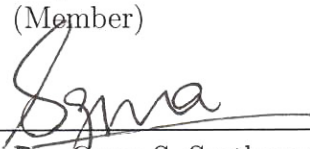
Approved:

  
\_\_\_\_\_  
Dr. Yong C. Kim  
(Chairman)

5 MAR 08  
date

  
\_\_\_\_\_  
Maj LaVern A. Starman, Ph.D  
(Member)

5 Mar 08  
date

  
\_\_\_\_\_  
Dr. Guna S. Seetharaman  
(Member)

5 Mar 08  
date

*Abstract*

As Field Programmable Gate Arrays (FPGAs) become more widely used, security concerns have been raised regarding FPGA use for cryptographic, sensitive, or proprietary data. Storing or implementing proprietary code and designs on FPGAs could result in the compromise of sensitive information if the FPGA device was physically relinquished or remotely accessible to adversaries seeking to obtain the information. Although multiple defensive measures have been implemented (and overcome), the possibility exists to create a secure design through the implementation of polymorphic Dynamically Reconfigurable FPGA (DRFPGA) circuits. Using polymorphic DRFPGAs removes the static attributes from their design; thus, substantially increasing the difficulty of successful adversarial reverse-engineering attacks. A variety of dynamically reconfigurable methodologies exist for implementation that challenge designers in the reconfigurable technology field.

A Hardware Description Language (HDL) DRFPGA model is presented for use in security applications. The Very High Speed Integrated Circuit HDL (VHSIC) language was chosen to take advantage of its capabilities, which are well suited to the current research. Additionally, algorithms that explicitly support granular autonomous reconfiguration have been developed and implemented on the DRFPGA as a means of protecting its designs. Documented testing validates the reconfiguration results and compares power usage, timing, and area estimates from a conventional and DRFPGA model.

## *Acknowledgements*

First and foremost, I owe all to my Lord and Savior, Jesus Christ, for His countless blessings and eternal salvation.

Next, my gratitude and thanks to my lovely wife and daughter for their understanding through sleepless nights and very full days. Without their selfless support, I would not be here today.

Thanks to my parents, sisters, and brother all scattered around the world...you never once tired of hearing my “adventure” stories no matter how many times I repeated them! Your support means so much to me.

A big thank you to my father-in-law for sharing his AFIT experience and some intense hours of poring over thesis edits...as usual, above and beyond.

To the VLSI crew, lots of thanks for the help and support in everything from VHDL questions to those late night discussions on the abstract and asinine topics that kept us from losing our minds.

Thanks to our sponsors for the support and direction. They are a truly critical link in bringing academic ideas to the operational world. Your continued advice and guidance will determine the future of our Air Force!

Thank you as well to the AFIT professors that have guided and evaluated my learning through the last 18 months. My time here has exposed me to a wealth of knowledge and expertise. The lessons learned will be applied throughout my career and in everything I do.

Finally, thanks to my advisor for his passion and understanding. An expert in his field and in assisting to shape my work into a presentable, polished form. You’ve spent some long days here in support of this research and for that I am grateful.

Samuel J. Stone

# *Table of Contents*

	Page
Abstract . . . . .	iv
Acknowledgements . . . . .	v
List of Figures . . . . .	ix
List of Tables . . . . .	xi
List of Abbreviations . . . . .	xii
 I. Introduction . . . . .	 1
1.1 Problem Statement . . . . .	2
1.2 Goals . . . . .	3
1.3 Scope and Assumptions . . . . .	4
1.4 Methodology . . . . .	4
1.5 Materials and Equipment . . . . .	4
1.6 Overview . . . . .	5
 II. Background . . . . .	 6
2.1 FPGA Background . . . . .	6
2.1.1 FPGA Architecture Overview . . . . .	6
2.1.2 Bitstream Overview . . . . .	9
2.1.3 JTAG Programming Interface . . . . .	9
2.2 Vulnerabilities Affecting FPGAs . . . . .	11
2.2.1 Vulnerabilities Overview . . . . .	11
2.2.2 Bitstream Interception . . . . .	12
2.2.3 Fault Injection . . . . .	14
2.2.4 Passive Circuit Analysis . . . . .	16
2.2.5 Altered Bitstream Attacks . . . . .	18
2.2.6 Bitstream Readback . . . . .	18
2.2.7 Vulnerabilities Summary . . . . .	19
2.3 Dynamic Reconfiguration . . . . .	20
2.3.1 Dynamic Reconfiguration Overview . . . . .	20
2.3.2 Dynamically Reconfigurable Designs . . . . .	22
2.3.3 Dynamic Reconfiguration Challenges . . . . .	24
2.4 Polymorphic Reconfiguration . . . . .	25
2.4.1 Polymorphic Reconfiguration Overview . . . . .	25
2.4.2 Polymorphic Reconfigurable FPGA Designs . . . . .	26

	Page
2.4.3 Polymorphic Reconfiguration Challenges . . . . .	27
2.4.4 Polymorphic Reconfiguration Summary . . . . .	27
III. Research Methodology . . . . .	29
3.1 Problem Definition . . . . .	29
3.1.1 Goals and Hypothesis . . . . .	29
3.1.2 Approach . . . . .	30
3.2 Research Boundaries . . . . .	32
3.3 System Services . . . . .	35
3.4 Workload . . . . .	35
3.5 Performance Metrics . . . . .	36
3.6 Parameters . . . . .	37
3.6.1 SUT Parameters . . . . .	37
3.6.2 Workload parameters . . . . .	39
3.7 Factors . . . . .	40
3.8 Evaluation Technique . . . . .	41
3.9 Experimental Design . . . . .	42
3.10 Methodology Summary . . . . .	43
3.11 Custom FPGA Design . . . . .	44
3.11.1 CLB . . . . .	45
3.11.2 Control Registers . . . . .	46
3.11.3 FPGA Data Network . . . . .	46
3.11.4 CLB Programming Network . . . . .	47
3.11.5 LUT Inversion Network . . . . .	48
3.11.6 DRFPGA Programming . . . . .	48
3.12 Reconfiguration Algorithm . . . . .	48
3.12.1 LUT Inversion Algorithm . . . . .	50
3.12.2 Functional Replacement Algorithm . . . . .	53
3.12.3 Obfuscation Mask . . . . .	56
3.13 Summary . . . . .	57
IV. Test Process and Results . . . . .	58
4.1 Test Circuits . . . . .	58
4.1.1 Test Circuit Overview . . . . .	58
4.1.2 Test Circuit Creation . . . . .	59
4.1.3 Adder . . . . .	59
4.1.4 Counter . . . . .	60
4.1.5 Comparator . . . . .	61
4.2 Reconfiguration . . . . .	63
4.2.1 Reconfiguration Overview . . . . .	63

	Page
4.2.2 Adder Reconfiguration . . . . .	65
4.2.3 Counter Reconfiguration . . . . .	66
4.2.4 Comparator Reconfiguration . . . . .	68
4.3 Test Results . . . . .	68
4.3.1 Test Results Overview . . . . .	68
4.3.2 Circuit Operation and Performance . . . . .	69
4.3.3 DRFPGA Programming Network Performance . . . . .	73
4.4 Summary . . . . .	78
V. Analysis and Conclusions . . . . .	79
5.1 Bitstream Protection . . . . .	79
5.2 Reverse Engineering Defense . . . . .	82
5.3 Circuit Protection Effectiveness . . . . .	84
5.3.1 Bitstream Interception . . . . .	84
5.3.2 Fault Injection . . . . .	84
5.3.3 Passive Circuit Analysis . . . . .	85
5.3.4 Altered Bitstream Attack . . . . .	87
5.3.5 Bitstream Readback . . . . .	87
5.4 Summary . . . . .	88
VI. Future Research . . . . .	89
6.1 Autonomous Secure DRFPGA Platform . . . . .	89
6.2 Biological Computing . . . . .	89
6.3 Reliability and Self-healing . . . . .	90
6.4 Defragmentation and Self-Optimizing . . . . .	91
6.5 Summary . . . . .	91
Appendix A. CLB Schematic . . . . .	92
Appendix B. LUT Inversion Algorithm . . . . .	93
Appendix C. Test Circuit LUT Contents . . . . .	100
Appendix D. Test Circuit Schematics . . . . .	106
Appendix E. JTAG Programming Network Example . . . . .	111
Appendix F. CLB Addressable Programming Network Example . . . . .	113
Bibliography . . . . .	115
Vita . . . . .	120

## *List of Figures*

Figure		Page
2.1.	Simplified FPGA structure demonstrating the fundamental components common amongst FPGAs. . . . .	7
2.2.	General JTAG boundary cell schematic. . . . .	10
2.3.	Results of fault injection on the successful completion of a basic AES encryption algorithm. . . . .	15
2.4.	SRAM array as viewed through Wentworth manual prober. . .	16
2.5.	VLIW Reconfigurable Processor. . . . .	21
2.6.	An abstract diagram highlighting evolutionary (left side) and traditional (right side) digital circuit design approaches [23]. .	26
3.1.	Abstract representation of the Reconfigurable FPGA as the SUT.	32
3.2.	VHDL CLB model diagram. . . . .	45
3.3.	Simple demonstration of the bubble pushing concept on a gate level circuit schematic. . . . .	50
3.4.	Simple demonstration of the pushing bubbles concept as applied to the LUT circuit schematic. . . . .	51
4.1.	Functional configuration diagram for n bit ripple carry adder. .	60
4.2.	Functional configuration diagram for an n-bit counter. . . . .	61
4.3.	Gate level implementation of a simple binary comparator. . . .	62
4.4.	Functional diagram demonstrating the hierarchical structure of a binary comparator. . . . .	63
4.5.	Demonstration of the counter output errors due to operation of the circuit during the reconfiguration process. . . . .	67
4.6.	Demonstration of the proper method to reconfigure a sequential circuit. . . . .	67
4.7.	Graph of the power usage for JTAG, CLB addressable network, and LUT inversion network . . . . .	75
4.8.	Graph of the clock cycles required for JTAG, CLB addressable network, and LUT inversion network reconfiguration . . . . .	77

Figure		Page
5.1.	Demonstration of the phased defense method of obfuscating the bitstream. . . . .	81
5.2.	Demonstration of the functional replacement method of obfuscating the input and output of a given circuit. . . . .	82
A.1.	Schematic of a CLB as based on the schematics of the <b>Xilinx® Virtex4®</b> FPGA [2]. . . . .	92
B.1.	Sample circuit for LUT inversion demonstration. . . . .	95
B.2.	Demonstration of steps one and two of the LUT inversion algorithm. . . . .	96
B.3.	Demonstration of steps three and four of the LUT inversion algorithm. . . . .	97
B.4.	The final product of the LUT inversion algorithm. . . . .	98
B.5.	Test waveforms demonstrating the application of the LUT inversion algorithm on a VHDL LUT circuit model. . . . .	99
D.1.	Test circuit for 8-bit adder. . . . .	107
D.2.	Test circuit for 8-bit counter. . . . .	108
D.3.	Test circuit for 8-bit comparator. . . . .	109
D.4.	Test circuit for 8-bit comparator circuit used for functional replacement. . . . .	110
E.1.	An example of an 8 node JTAG Network. . . . .	111
F.1.	An example of an 16 node CLB addressable network. . . . .	113

## *List of Tables*

Table		Page
2.1.	Truth table representation of two input AND gate. . . . .	8
3.1.	Potential outcomes of SUT . . . . .	35
3.2.	Experimental input counts. . . . .	43
4.1.	Specifications for the test circuit pre and post reconfiguration. .	71
4.2.	Area results for reconfiguration hardware. . . . .	76
C.1.	Ripple Carry Adder LUT Values. . . . .	100
C.2.	Counter LUT Values. . . . .	101
C.3.	Comparator LUT Values for stage 1 of the comparator operation.	102
C.4.	Comparator LUT Values for stage 2 of the comparator operation.	103
C.5.	Comparator LUT Values for the functional replacement com- parator, phase 1. . . . .	104
C.6.	Comparator LUT Values. . . . .	105

## *List of Abbreviations*

Abbreviation		Page
FPGA	Field Programmable Gate Array . . . . .	iv
DRFPGA	Dynamically Reconfigurable FPGA . . . . .	iv
HDL	Hardware Description Language . . . . .	iv
VHSIC	Very High Speed Integrated Circuit HDL . . . . .	iv
CT	Critical Technology . . . . .	1
ASIC	Application Specific Integrated Circuit . . . . .	6
SRAM	Static Random Access Memory . . . . .	7
LUT	Look Up Table . . . . .	7
JTAG	Joint Test Action Group . . . . .	9
ROM	Read Only Memory . . . . .	13
FIPS	Federal Information Processing Standards Publication . .	13
PLA	Programmable Logic Array . . . . .	20
SUT	System Under Test . . . . .	32
CUT	Component Under Test . . . . .	33
LFSR	Linear Feedback Shift Register . . . . .	36
TMR	Triple Modular Redundancy . . . . .	90

# ANTI-TAMPER METHOD FOR FIELD PROGRAMMABLE GATE ARRAYS THROUGH DYNAMIC RECONFIGURATION AND DECOY CIRCUITS

## I. Introduction

The United States (US) military maintains military dominance because of the quality of its members, training, and technological superiority. Due to the relatively low cost of exploitation and abundance of reverse engineering tools and methods, the technology used by the US military is a common target for adversaries. Technology acquisition through subversive means and analysis/reverse engineering of US military Critical Technologies (CT) represents a significant threat to the US military's technological lead over adversaries. In order to counter this threat, it is essential that the Department of Defense (DoD) and the United States Air Force (USAF) implement procedures to safeguard proprietary technology.

It is not enough to assume that current safeguards will keep DoD equipment and CTs out of adversarial hands. A US Government Accountability Office (GAO) report published in 2007 describes a critical shortcoming in the DoD's ability to maintain control of its equipment. Over 190,000 weapons meant to be distributed to the Iraqi Security Forces are unaccounted for [11]. These weapons not only represent million of dollars in DoD funds, but also US military technologies that could be in possession of entities we are facing in combat. Without additional safeguards the DoD could lose the technological edge gained through research and development of CTs.

One of the relatively new technologies seeing increasing use in the USAF is the Field Programmable Gate Array (FPGA). A particularly critical vulnerability inherent in the current FPGA platforms is the ease of reverse engineering. Because FPGAs contain flexible architecture defined by stored configuration data, circuit functions

implemented on the devices are susceptible to theft and tampering. Consequently, adversaries may be able to identify and reverse engineer the functionality of FPGAs used in military applications. In addition to reverse engineering, there exist malicious entities interested in tampering with FPGA circuits; thus, altering their operation. If an FPGA device is unaccounted for, as was the case for the 190,000 weapons in Iraq, then any technology associated with the device can be considered compromised. Additionally, deployed FPGA systems are not secured against tampering and cannot be considered safe for use in critical applications. Countermeasures for threats against military FPGA designs must be investigated and implemented to secure FPGA designs and CTs.

FPGAs offer substantial potential for dynamic reconfiguration by harnessing the strengths of the FPGA architecture, but current technology is not sufficient to protect DoD CTs against theft and malicious misuse. Someone must address the crucial need for CT protection to maintain the DoD's technological lead over adversaries. This research analyzed current FPGA architectures and proposes a new enhanced anti-tampering capable architecture fulfilling the need for a method of securing CTs. A particular focus is the use of dynamic reconfiguration to defend against adversarial tamper and reverse engineering attacks. The ability to create decoy circuits on the FPGA is explored and demonstrated. The proposed methods are included in the simulated architecture design and verified through synthesis. The final design incorporates dynamic programming and decoy circuits to secure FPGA implemented CTs from adversarial theft.

### ***1.1 Problem Statement***

Current FPGA architecture does not provide adequate protection against adversarial tampering or reverse-engineering attacks targeting DoD technology. There exists ample documentation on the vulnerabilities affecting FPGAs to allow adversaries with relatively modest experience and/or resources to successfully obtain FPGA implemented designs or even alter programs to operate in a manner inconsistent with

the intended application. While defensive technologies exist on modern designs, these do not address all methods used by malicious adversaries. Additionally, capabilities in the promising dynamic configuration field fall short of the required granularity, efficiency, and overall effectiveness for adequate defence against theft and tampering.

## **1.2 Goals**

This research directly supports the AFRL/RYT's mission of developing technologies to prevent or delay the exploitation of critical program information. In a broader sense, this research supports the Air Force's air superiority mission through safeguarding of critical technological intellectual property; so that weapon system capabilities stay out of adversarial hands. This research extends beyond the USAF mission to support the nation's technological edge and its research and development investment protection by preventing or delaying the exploitation of critical technology. The desired outcome is a new architecture and supporting dynamic configuration methods that can be implemented in USAF weapons systems at a reasonable cost in terms of development, manufacturing, and performance degradation.

Hardware design is only one aspect to consider in developing a secure FPGA design platform. In order to fully take advantage of the custom design, software algorithms use the capabilities of the secure hardware platform. Algorithms using the hardware must leverage the benefits of dynamic reconfiguration and the reconfiguration hardware must provide adequate granularity. This thesis investigates the total solution methodology.

Finally, the hardware and algorithm product must compare favorably to current designs in operation speed, power usage, and size. This research fully compares viable options while realizing a synthesizable implementation.

### ***1.3 Scope and Assumptions***

The research is limited to the hardware contained within a Xilinx® hardware model coded in the VHDL programming language. While the model is simulated and synthesized, actual fabrication or implementation on an FPGA does not fall within the scope of this research. All research, tests, and results will be limited to digital logic simulations/synthesis of components contained within the actual FPGA hardware. External factors (i.e. heat, power limitations) will not be considered.

### ***1.4 Methodology***

The foundation for this research is the implementation and evaluation of a Virtex4® based FPGA device modified to provide dynamic reconfiguration capabilities beyond the existing platforms. The dynamic reconfiguration supports the execution of defensive measures to include design and bitstream obfuscation. This research proposes two reconfiguration strategies for securing circuits using the custom hardware design; functional replacement and LUT inversion. While both strategies serve to protect the design implemented on the proposed FPGA platform, they employ unique aspects and granularity of dynamic reconfiguration to address different vulnerabilities.

### ***1.5 Materials and Equipment***

The following software programs were used in the modeling, testing, and verification of the proposed design:

**Mentor Graphics Modelsim® SE 6.2e** Allows VHDL file editing and compiling.

Performs VHDL hardware model testing via simulation.

**Altera® Quartus® II v6.0** Performs VHDL hardware model interconnection. Abstracts VHDL code to visual diagrams.

**Xilinx® ISE<sup>TM</sup> 9.2i** Xilinx® Performs synthesis, design, and verification of implementation on Xilinx® FPGA devices. Also provides parametric estimates for operating speed, power, and area requirements for the targeted Xilinx® device.

**ActivePerl v5.8.8** Executes FPGA bitstream creation scripts.

## **1.6 Overview**

This document is organized into 6 different chapters. Chapter 1 is the introduction to the problem statement and approach. Chapter 2 provides background on the vulnerabilities affecting current FPGA designs. Chapter 3 outlines the approach to solving the problem and testing the method applied. Chapter 4 consolidates the results of the implementation and testing phase outlined in Chapter 3. Chapter 5 summarizes the analysis and conclusions based on Chapter 4's test results. Finally, Chapter 6 closes with ideas on future research based on the insights learned from this research effort.

## II. Background

The following chapter serves to familiarize readers with FPGA designs and their vulnerabilities. There exist various articles documenting vulnerabilities affecting FPGAs and Integrated Circuits (ICs). While this chapter should not be considered an all encompassing study of FPGA security, it does serve as an introduction to the critical dangers facing FPGA designers and users.

This chapter also serves to introduce the concepts behind polymorphic dynamic configuration; specifically, how it relates to the security of FPGA designs. The reader is presented with background information on FPGAs and reconfiguration followed by examples of different dynamic reconfiguration implementations. Finally, potential challenges and shortcomings in the field of dynamic reconfiguration are explored.

### 2.1 *FPGA Background*

*2.1.1 FPGA Architecture Overview.* As integrated circuit technology has progressed so has the desire for organizations to create and test their own integrated circuit designs. Creating a custom integrated circuit in the traditional sense is expensive and the cost per device is prohibitive for cases where a designer is not seeking to mass manufacture devices. Although Application Specific Integrated Circuit (ASIC) devices have provided a relatively low cost method to manufacture low volume designs for over 25 years, FPGAs have presented an even more flexible way to design, create, and test custom integrated circuit designs with relatively modest up-front investment capital.

The primary advantage to using FPGAs is their flexibility and ability to be reprogrammed. FPGAs are used to implement digital functions in hardware. The functions are programmed in an HDL such as VHDL or Verilog and saved to a file. This HDL code file is compiled and downloaded to the FPGA device for implementation. A similar process can be used to create ASICs with the exception that ASICs require fabrication at specialized facilities. The FPGA structure allows it to be programmed with low initial investment and FPGAs can be easily reprogrammed with

new or updated designs. These attributes make FPGAs a well-suited platform for low-volume designs as well as digital circuit prototypes.

One of the lead developers in the FPGA field is **Xilinx®**. **Xilinx®** FPGAs have been the platform of choice for dynamic reconfiguration implementations based on their built-in reconfiguration capabilities [10,16,28,30,33,45]. The **Xilinx® Virtex4®** Static Random Access Memory(SRAM) based FPGA was chosen as the base for the research model in order to provide a capable foundation for dynamic reconfiguration work. The research into vulnerabilities is relevant to SRAM based FPGAs in general, not only the **Virtex4®**.

In general, FPGAs implement user designs through the interconnecting and configuration of Configurable Logic Blocks (CLBs). Each CLB is formatted to implement a small portion of the design using components such as one or more flip-flops, Look-Up Tables (LUT), and data storage elements. The equivalent component may have different names for varying FPGA manufacturers; for example, Altera refers to their CLB equivalent components as Logic Elements (LEs). CLBs are interconnected by a data network to create the overall design of the circuit. Figure 2.1 shows a simplified pictorial view of an FPGA.

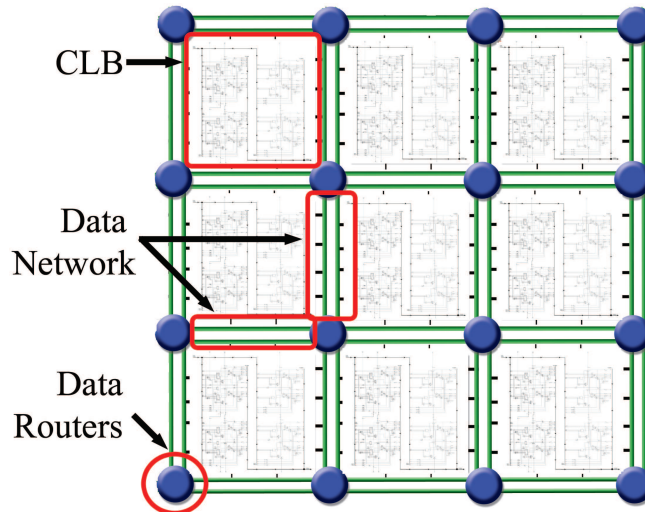


Figure 2.1: Simplified FPGA Structure. The fundamental components (CLBs) are interconnected via the data network and data routers. While this description is oversimplified, it captures the basic understanding necessary for discussion.

FPGA devices can be decomposed into the following components. The nomenclature is **Xilinx**<sup>®</sup> specific because of the **Virtex4**<sup>®</sup> based model used in this research.

**LUT** LUTs contain the fundamental functions of the circuit. They function similar to a truth table in that values are stored within the LUT at locations addressed by the inputs. A truth table representing a two input AND gate function is demonstrated in Table 2.1. Some LUTs contain shift register, memory, or other functions in addition to the ability to implement a truth table.

**Slice** Each slice contains two LUTs for implementing digital circuits. The slice also contains routing hardware to select input and output locations and FFs for implementing sequential designs.

**CLB** The CLB is the largest functional component within the FPGA. Each CLB contains 4 slices: two L-Slices and two M-Slices. The M-Slices contain data storage and shift components in addition to the LUT functions and routing hardware.

**Data Network** The data network routes data between the CLBs to implement complex functions.

**Bitstream** The bitstream is the program stored and implemented on the FPGA. It is compiled off-chip and downloaded to the FPGA.

Table 2.1: A truth table representation of a two input AND gate. The output value is selected based on the two inputs as in a logic truth table. A LUT can be viewed as a hardware implementation of a truth table.

Two Input AND Gate Truth Table		
Input A	Input B	Output Value
0	0	0
0	1	0
1	0	0
1	1	1

*2.1.2 Bitstream Overview.* FPGAs are configured to match user specifications. Unlike ASICs, which are configured in an ASIC facility and cannot be reprogrammed, FPGAs can be reprogrammed using a computer with the appropriate software. Once the user design is created, using either VHDL programming software or other specific software for the FPGA, the design is compiled and saved to the bitstream file. The bitstream contains all configuration data necessary to create the user's design on an FPGA, including the bus routing between CLBs and the format for every CLB to be used. The bitstream can be saved for implementation in the future and is the single file needed to recreate the user designs on an FPGA. The bitstream must be downloaded to the FPGA either from a computer or from a storage medium such as EPROM/PROM/ROM connected to the FPGA. During the transmission from the source to the FPGA itself, the bitstream is encrypted on certain FPGAs (including the **Xilinx® Virtex4®**) to prevent capture of the information from the transmission lines. For example, on the **Virtex4®** the bitstream is encrypted before transmission to the FPGA by the software package used to generate the bitstream. The encrypted bitstream is then transmitted to the FPGA, where it is decrypted for use in configuration [1]. Once the bitstream is downloaded onto the local FPGA storage, the configuration takes place on the FPGA itself. While the bitstream may be considered the heart of the FPGA, it is also a vulnerable point for acquisition of the FPGA design. As such, it must be protected to prevent theft of FPGA designs.

*2.1.3 JTAG Programming Interface.* Most current FPGA designs use the Joint Test Action Group (JTAG) boundary scan standard to load the bitstream onto the FPGA. The boundary scan standard is a serial, sequential method of loading data into a device for programming or testing. The fundamental component of the boundary scan system is the boundary scan cell. Figure 2.2 depicts a generalized schematic of the cell. The cells are arranged in a chain with the *To Next Cell* line output feeding the *From Previous Cell* input on the next cell. This arrangement

permits the serial loading of test and/or configuration vectors to a device. Each cell can operate as a shift register, accept new data on the *From Previous Cell* line, and provide current data out on the *To Next Cell* line. As long as the ClockDR clock is operating and the ShiftDR line is high, the boundary scan cells will continue to shift data in serially, one bit per clock cycle from the first cell's *From Previous Cell* line. Using this method, an entire FPGA bitstream can be loaded onto the device through a single bit input. Some FPGAs, including the Xilinx® Virtex4®, also include functionality to load a single column of CLBs at a time by bypassing portions of the boundary scan cell chain.

Once the entire chain or the column currently under configuration contains the configuration bits within the ClockDR fed FFs, the ClockDR signal is halted and the UpdateDR signal is cycled through one clock cycle while the Mode signal is high. This process loads a single bit into the selected CLBs in parallel through the simultaneous loading of every UpdateDR FF. In order to program the FPGA or a portion of the FPGA, the entire process must be repeated for each bit of the bitstream. The ClockDR and UpdateDR clocks are able to run at a significantly faster clock rate than the general system clock since there are limited gates between FFs to increase delay. See Appendix E for an example of a JTAG programming implementation.

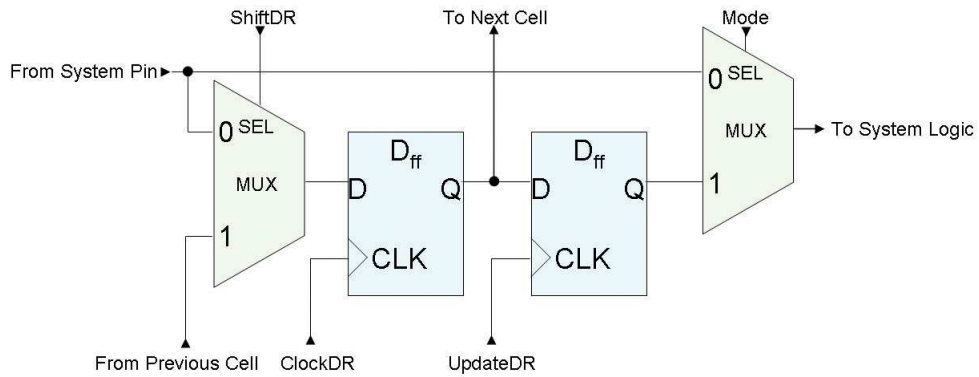


Figure 2.2: General schematic of boundary scan cell based on example presented by Bushnell and Agrawal. [34].

## 2.2 *Vulnerabilities Affecting FPGAs*

*2.2.1 Vulnerabilities Overview.* One of the concerns surrounding the use of FPGAs is the insecurity of the device itself [46]. The configurability of the FPGA is one of the reasons it is inherently more insecure than standard VLSI ICs. Although there are numerous papers and studies on a wide variety of FPGA vulnerabilities, many of the attacks designed to acquire circuits stored on an FPGA focus on the bitstream. As the bitstream contains all of the necessary information to create the FPGA design, anyone that is able to compromise the bitstream information would have complete ability to recreate the design as many times as desired or alter the design to meet their specifications. If the bitstream is not properly protected, it represents a significant vector for proprietary design theft. Although there are steps in place to protect the bitstream, every protective measure comes under attack as more FPGAs are used for sensitive design implementation. In addition to bitstream attacks, which are particularly well suited for compromising the circuit design, another focus area is passive circuit analysis. This can be performed either by capturing information at the circuit boundaries (inputs and outputs) as a non-invasive procedure or by actually stripping the physical layers of protection from the FPGA and monitoring the hardware. These attack methods are detailed later, but it is important to understand the methods more likely to be employed by aggressors. The following FPGA vulnerabilities represent critical vectors for design theft and tampering:

**Bitstream Interception** Adversaries can capture the bitstream during transmission from an off-chip location to the FPGA. The bitstream in current FPGA platforms is encrypted during transmission, but an adversary could leverage other attacks listed to weaken the encryption and obtain the design.

**Fault Injection** The goal of fault injection attacks is to provide input or power values beyond the tolerance of the FPGA hardware. Examples include adjusting the power supply, clock signal, or inputs to cause the circuit to detour from the intended operation. Fault injection can be used to alter the design functional-

ity, cause the circuit to cease functioning completely, or circumvent defensive functions protecting the bitstream or other critical data.

**Passive Circuit Analysis** In performance of this attack, adversaries seek to monitor the circuit operation without impacting it. They may capture input/output combinations, thermal signatures, or inspect individual transistor values to gain useful information. The information obtained can be used directly to compromise the device or aid in other attacks.

**Altered Bitstream Attack** Adversaries can modify the bitstream during transmission to influence the FPGA device operation.

**Bitstream Readback** FPGAs contain functionality enabling them to provide the bitstream data via the JTAG interface. There is a security bit in place for Xilinx® Virtex4® FPGAs to prevent this activity, but other attacks could be used to bypass the security bit.

*2.2.2 Bitstream Interception.* As mentioned in the bitstream overview, the bitstream is transmitted to the FPGA device during the configuration stage. The FPGAs addressed in this thesis (SRAM FPGAs) must be programmed before use. Also, it is important to note that due to the volatile nature of the SRAM configuration cells, the FPGA must be reprogrammed whenever it loses power. It is possible to intercept the transmission of the bitstream and capture the information for later analysis during the configuration stages given the off-chip location of the configuration bitstream. This method is not easily executed as the bitstream is transmitted as an encrypted file in the current FPGA devices.

Although the bitstream is an important and vulnerable feature of the FPGA, it was not until 2000/2001 that a mainstream FPGA manufacturer introduced substantial security features on an SRAM FPGA board: the Virtex II® FPGA by Xilinx® [26]. Previous to the implementation of encrypted data transfer, the most efficient means of protecting sensitive information was to program the FPGA in a secure location and place the FPGA device on battery backup to keep the volatile memory in

the FPGA from losing its configuration [26]. While this method avoids capture of the bitstream by adversaries, it increases the cost of design due to the need to provide battery backup for the FPGA device. In addition, it complicates the reconfiguration process and does not address remote reconfiguration operation vulnerabilities.

In an effort to address the need for more practical means of protecting sensitive design information, the **Xilinx® Virtex4®** FPGAs utilize encryption during the transmission of the bitstream to the FPGA. In cases where the storage medium resides on the same device as the FPGA (i.e. in Read Only Memory (ROM) on the board) or in cases where the bitstream is transmitted to a remote device, encryption of the bitstream is essential to protect the design. Early **Xilinx®** FPGAs used the Data Encryption Standard (DES) encryption algorithm. DES, which was created in the late 70s, has become obsolete due to its relatively short key (56 bits) and the development of the Advanced Encryption Standard (AES). A DES encrypted message can be theoretically cracked in as little as 3.5 minutes using substantial computing power and investment (on the order of \$10MIL). While \$10MIL investments may be affordable for large organizations and governments, smaller organizations could crack the DES encryption algorithm in just 2.5 days with as little as \$10K invested [44]. Users with sensitive designs should implement them on FPGAs using the AES standard of encryption which protects against brute-force bitstream attacks.

The **Virtex4®** FPGA uses the AES standard, which is the current encryption standard as described by Federal Information Processing Standards (FIPS) Publication 197 [35]. While this does provide substantial protection due to its increased key length (at least 128-bits) and improved algorithm, there is still the possibility of compromise. The key used to decrypt the bitstream must be stored on the actual FPGA. If an adversary could successfully obtain the key from the FPGA memory, then it would be a trivial matter to decrypt the messages. Therefore, it is crucial FPGA designers take special care to protect the region of the FPGA device housing the encryption key. The physical defenses against value inspection are covered in the physical inspection vulnerabilities section of this paper.

*2.2.3 Fault Injection.* Fault injection is a generic term describing the injection of faults into digital systems using a variety of attacks including voltage higher or lower than system tolerances, voltage spikes, or clock glitches. An attacker may use any of these methods to cause the system to malfunction with intentions to reveal information useful in further attacks. In the case of bitstream vulnerabilities, fault injection has the most potential with regards to interrupting the encryption of the bitstream or forcing the FPGA to reveal the bitstream through its readback functionality.

As stated previously, a primary defense against bitstream theft/reverse engineering is the encryption of the bitstream. Using the AES encryption standard will render brute force bitstream decryption unfeasible. However, the encryption strength relies on the successful completion of the encryption process. Attackers with access to the FPGA and storage medium could use fault injection techniques to interrupt the bitstream encryption and create a partially encrypted bitstream decipherable in a more reasonable time than a correctly encrypted bitstream. Therefore, the goal of an adversary is no longer to break the encryption of the bitstream using brute-force methods, but rather to interrupt the successful encryption/decryption process.

The AES encryption implemented on the **Virtex4**<sup>®</sup> FPGA relies on successive iterations of encryption and decryption for the creation of a valid encrypted bitstream. Bertoni, et. al. discuss the impact single and multiple fault injections have on hardware AES encryption processes [13]. Although the tests targeted AES encryption, the results are pertinent for any iterative encryption process. The core AES encryption algorithm was found to be susceptible to even single fault injections. The AES algorithm illustrated in Figure 2.3 reveals faults injected at early rounds of the encryption process resulted in a significantly altered output. In fact, faults injected as late as the eighth (out of 10) stage resulted in a 50% error rate for the encrypted message.

While an incorrectly encrypted message will not immediately allow an attacker to obtain a protected design, the tests highlight the susceptibility of the AES algorithm

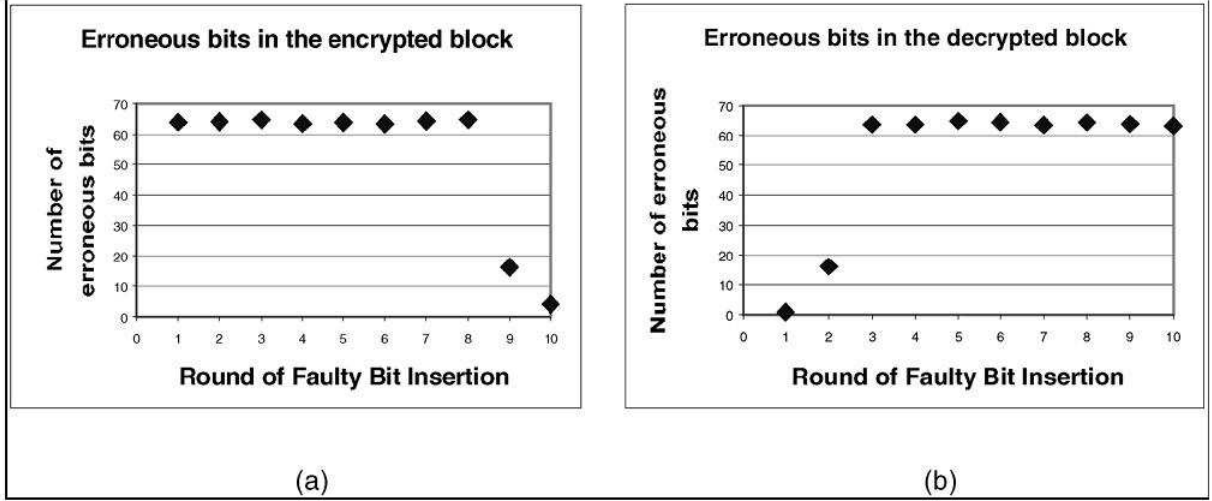


Figure 2.3: Results of fault injection on the successful completion of a basic AES encryption algorithm. Figure 2.a. indicates that faults injected early in the encryption process result in the highest percentage of errors in the encrypted message (about 50%). Figure 2.b displays the inverse is true for the decryption of a message. [13].

to fault injection. At the very least, the FPGA design will operate in an unknown, faulty manner, making it possible for a clever, dedicated adversary to use the results to obtain the security key and/or the design itself. It is important that the encryption process is supplemented to limit the impact of faults. Unfortunately, many of the generalized protection methods may not translate to the FPGA design due to limited space and computational ability. Perhaps the most appropriate protection method is based on redundancy. By implementing a decryption block following the encryption, the encrypted message can be validated. Implementation of the decryption hardware will approximately double the total area and power requirements due to similarities between the encryption and decryption processes [13]. Therefore, the redundancy technique may not be applicable for limited resource implementations.

The encryption disruption vulnerability is just one example of how fault injection can be used to circumvent the FPGA design protection methods. Countless opportunities exist for curious or malicious individuals practicing fault injection techniques against FPGA bitstreams. Redundancy is one method of protecting against fault injection, but Skorobogatov and Anderson also discuss a method of fault injection

tion protection using a “dual-rail” technology [13]. Using this technology, a value is based on a combination of two lines. For instance, instead of a signal (logical one or zero) being represented by a single line, the signal would be represented by two lines having opposite values (i.e. one tied to 5v and the other to ground). Two signal lines will limit the susceptibility of the circuit to single transistor failures and, according to the authors, also make the circuit more resistant to power analysis attacks. The dual rail design significantly increases the area and power consumed by the data routing hardware. Widespread use of the design within a circuit would also require substantial changes in the hardware to accommodate the two value interpretation.

*2.2.4 Passive Circuit Analysis.* Although the circuits contained within FPGA IC packages are far too small for unaided visual analysis, numerous tools exist that enable viewing of the circuits and even individual transistors, as demonstrated in Figure 2.4.

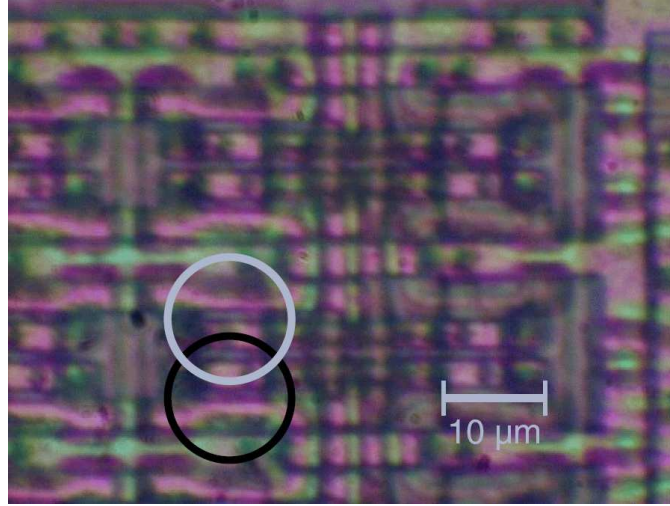


Figure 2.4: SRAM array as viewed through Wentworth Labs MP-901 manual prober. By illuminating the black circle with a photographer’s flash, Sergei P. Skorobogatov and Ross J. Anderson were able to store a ‘1’ in the SRAM cell. Alternatively, by illuminating the region within the white circle, they were able to store a ‘0’ in the cell. In this way, the authors are able to combine Analysis and Fault Injection techniques [13].

By viewing transistors and patterns within the design, attackers can identify key portions of a design and target them. Passive analysis is best used in conjunction with other methods of tampering with the design. This form of analysis, when properly carried out, can provide an attacker with valuable information with which further means of identifying and compromising sensitive design information can be performed. The **Xilinx**® FPGAs, like most ICs, have the most rudimentary of defense mechanisms: they are housed within an IC “package” with an epoxy body covering the circuit itself. The standard epoxy package housing of most IC designs will not deter a dedicated attacker from gaining access to the design, but serves mainly to deter casual investigation or accidental damage to the circuit hardware. **Xilinx**® has taken the extra step to cover especially critical portions of their design (i.e. the key for bitstream encryption) with multiple layers of metal; thus, making the de-packaging process that much more difficult [42]. This fact combined with the volatile nature of the key memory greatly reduces the chance of an attacker successfully obtaining the key. However, the physical barriers should not be considered sufficient to protect critical designs. Time, experience, and determination will allow an attacker to slowly strip away physical defences to reveal sensitive designs contained within.

Another popular method of passive analysis is power usage monitoring. By capturing and studying patterns in the power used by an FPGA, adversaries can map the timing for the chip processes. This information can be incredibly useful in carrying out fault injection or other attacks. Kamawal demonstrated that by viewing the power usage as a waveform, certain key points in an algorithm are visible [25]. As multiple data sets are captured, a composite signature for the FPGA operation can be constructed. Pattern analysis will highlight periods of increased activity, possibly the initial decryption period for an FPGA in the configuration cycle or recurring, critical tasks. As stated previously, Skorobogatov and Anderson discuss the idea that incorporating a dual rail logic design may protect against some power analysis attacks. Another method of protecting against power analysis is the random or pseudo-random reorganization of FPGA functional components. While there is not a large amount of

literature on the use of reconfiguration for power analysis defense, implementation of an automatically reconfiguring FPGA device will substantially hinder malicious power analysis attacks and defend against fault injection attempts. By creating a “moving target” the FPGA designer has taken the element of stability from the attacker and added an entirely new dimension of information obfuscation.

*2.2.5 Altered Bitstream Attacks.* It is important to maintain the confidentiality of sensitive or proprietary information stored on an FPGA. Although encryption of the bitstream may prevent an adversary from obtaining the design, someone with access to the transmission medium can alter the data flowing to a remotely configured FPGA or simply transmit a message of his choice. Attackers may be more interested in maliciously altering a bitstream downloaded to the FPGA than stealing the device’s design. While there has been adequate research on the other vulnerabilities, the altered bitstream vulnerability has not been substantially addressed in open research. One method of protecting against adversaries planting malicious designs on an FPGA is to use a combined encryption/authentication scheme as proposed by Parelkar and Gaj [36]. They propose utilizing the EAX functionality of the AES encryption algorithm to verify that the data received by the FPGA is valid and credible. Although two methods are presented in their paper, the benefit lies with the FIPS compliant method as it will more easily obtain accreditation through most agencies. The primary drawback to authentication implementation is the need for additional resources on the FPGA, which takes from the overall power/space available for the primary FPGA functions. If the space and power can be sacrificed, then it is highly recommended that some form of authentication be implemented especially for critical devices.

*2.2.6 Bitstream Readback.* While not technically a vulnerability, users implementing designs on FPGAs must be aware of the readback options available for the FPGA before fielding the design. The readback functionality is meant to provide users a way to obtain the bitstream directly from the FPGA typically through a

JTAG standard interface. The bitstream readback function and standard format for the connection make the readback function a perfect “first stop” for any adversary attempting to gain access to the bitstream. Xilinx® offers a simple security setting on the FPGA to prevent readback of the bitstream [48]. The Xilinx® documentation states that the only way to change the security setting once it has been set is to recycle the power or assert the *PROGRAM* input. Either of the actions will erase the configuration of the FPGA and require a new configuration stream to be loaded to the FPGA.

As stated in the fault injection section of this thesis, transistor values can be altered using optical radiation. This fact casts doubt upon the claim that the security setting cannot be altered without resetting the FPGA. Although research on this particular approach is not widespread, it is an important aspect of FPGA security that must be examined. At the very least, the security bit and readback function must be considered a potential security hole. Users should not completely rely on this functionality to protect critical designs.

*2.2.7 Vulnerabilities Summary.* As with all technology, FPGAs offer great possibilities, but misuse can seriously hinder any operations relying on the device and result in compromise of sensitive data. As FPGA use becomes more prevalent for proprietary designs, so has the overall community effort to discover ways to view/acquire those designs. While there are a variety of methods available to compromise the FPGA, the bitstream represents a tempting target for adversaries due to its fundamental place in the operation and programming of FPGAs. Organizations using FPGAs must be aware of the existing vulnerabilities and fund protective measure research to avoid operational impact due to design theft or sabotage. While there is no one end-all defensive measure for bitstream vulnerabilities, every method studied and implemented will further ensure that the mission will continue to succeed. Currently, the actual theft of the bitstream has not been considered as the primary vector of attack against FPGAs. Without a concerted community effort to discover more

defensive methods, the FPGA users will remain unaware of the true threat against their agency's operations.

### ***2.3 Dynamic Reconfiguration***

*2.3.1 Dynamic Reconfiguration Overview.* The primary advantage to the use of FPGAs versus other forms of custom circuits (i.e. ASIC, Programmable Logic Array (PLA), etc.) is the reconfigurable nature of the circuit. Traditionally, FPGAs are programmed using a serial or Universal Serial Bus (USB) interface. During the configuration process, the user downloads a design to the FPGA board using proprietary software. The software determines the placement, configuration, and routing of CLBs. This information is encoded in the bitstream and downloaded to the FPGA. Once the encoded bitstream is downloaded to the FPGA, the design is implemented. This method of programming FPGAs leaves the design vulnerable to multiple reverse engineering attacks. One method of protecting against such attacks is to implement a polymorphic DRFPGA.

Dynamic reconfiguration refers to the ability of a circuit to reconfigure itself while in operation. In order to reconfigure the FPGA while it is in operation, the user must be able to replace some, but not all, of the FPGA components specified by the bitstream information; thus, retaining the operating functions while replacing or creating additional functional components on the FPGA. The new design is implemented with little to no impact on the system output or operation. Dynamic reconfiguration offers no advantages if it imposes the same penalties as completely reconfiguring the device. Dynamic reconfiguration designs cannot only be optimized for current work loads, but can also retain the ability to perform additional specific tasks. One relatively common approach is to supplement a general purpose processor with specific computational modules. An example is Lodi et al.'s Very Long Instruction Word (VLIW) reconfigurable processor [7]. While a static core provides the majority of the computational functionality, a PiCoGa module is dynamically altered to meet current processing needs. This allows the processor to use custom designed al-

gorithms (i.e. Digital Signal Processing (DSP) programs) through the PiCoGa while also providing the capability to update the existing custom programs or replace them altogether for differing applications. Figure 2.5 depicts the relation between the static core and the reconfigurable PiCoGa module.

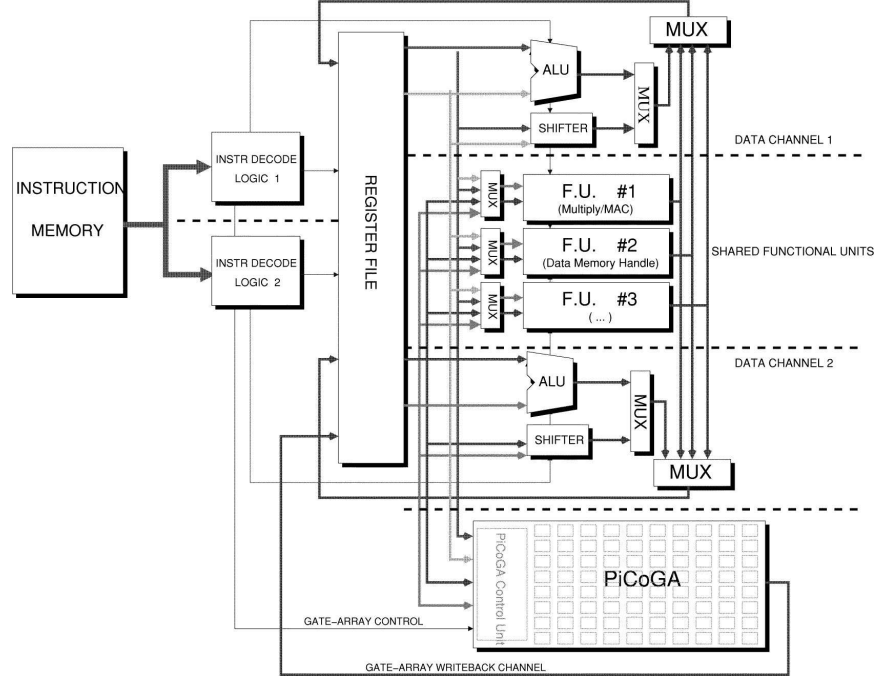


Figure 2.5: VLIW Reconfigurable Processor by Lodi et. al [7]. The PiCoGa is reconfigurable dynamically (while in operation) but the remaining modules are static.

Lodi et al. are not the only researchers to propose and implement a reconfigurable platform. In fact, multiple reconfigurable devices exist for FPGAs taking advantage of partially modifying the bitstream for a particular FPGA design; however, Kalte et al. discuss the lack of support and/or commercial programs supporting partial bitstream reconfiguration [16]. The research of Kalte et al. finds there were some non-commercial programs available for editing and downloading modified bitstreams (i. e. PARRBIT), but these tools are not able to fulfill the needs of an autonomous reconfiguring FPGA. Xilinx<sup>®</sup> also discusses limited reconfiguration potential for their Virtex II<sup>®</sup> line of FPGAs in Application Note 662 [47], but the reconfiguration is limited in scope to specific functions and does not contribute significantly to the dynamic reconfiguration solution. One short-coming is the lack of

fine granularity. Reconfiguration only applies to predefined modules and cannot be used to target arbitrary circuits/sub-circuits within the design. More focus is necessary on providing valid flexible solutions for users wishing to implement dynamic reconfigurable designs on FPGAs.

Polymorphism adds another aspect to the dynamic reconfigurable FPGA field. This thesis takes a polymorphic design to be one exhibiting granular reconfiguration and a substantial amount of autonomy when reconfiguring. This definition is not exact, but rather a descriptive outline serving to guide research direction and initiatives. While existing reconfigurable devices typically rely on predefined substitution of modules, polymorphism dictates an alteration of state, form, and function as defined in an algorithm. For instance, while the processor proposed by Lodi et al. is dynamically reconfigurable, it is not truly polymorphic because it simply replaces designated logic regions with different predefined functions. Another aspect of polymorphism is the inferred autonomy of polymorphic circuits. Polymorphic circuits are able to reconfigure themselves with minimal guidance in place via the reconfiguration algorithm. This aspect is discussed in depth as an evolutionary design method.

In studying the field of dynamic reconfiguration, it becomes evident that the primary focus has been reconfigurable devices that utilize preformed modules loaded from off-chip storage. Although this works well for the implementation of custom processors, it does not address the need for higher security through reconfiguration. In the next section, methods of dynamic reconfiguration specifically relevant to the security aspects of dynamic reconfiguration are discussed.

*2.3.2 Dynamically Reconfigurable Designs.* The DRFPGA field has seen multiple designs aimed at providing flexible, efficient solutions for users seeking to perform a range of tasks [12,21,32]. The current designs vary primarily in the method of reconfiguration rather than the motivation behind the reconfigurable nature of the devices. Although the designs may not be well suited to reconfiguration for

protection of circuit design, the methods themselves help to shed light on how dynamic reconfiguration can be achieved.

**Xilinx®** is a major player in the FPGA manufacturing field and has a commercial solution for customers seeking to implement a dynamically reconfigurable platform. As early as 2001, Wu et al. discussed the increasing attention gained by DRFPGAs and highlighted a **Xilinx®** approach to reconfiguring the FPGA using modified **Xilinx®** XC4000E CLBs [14]. While Wu et al.’s article focuses on an algorithm addressing the complex placement problem for DRFPGAs, it also highlights an interesting method of implementing the dynamic reconfiguration. When the reconfiguration process is initiated, the values for the current operations are stored in Micro Registers. The new configuration is stored as a “plane” of Configuration Memory Cells (CMCs) and is mapped over the existing CLB array. Once the change has been mapped and implemented, the operations resume using the values stored in the Micro Registers [14]. This method of reconfiguration has limited use in the security field due to the need for precompiled CMCs. While it may add a level of complexity to the progress of reverse-engineering, pattern analysis could reveal the limited number of CMC planes in use and may provide the attacker with enough information to accurately predict which planes will be implemented.

Lala and Walker present a very different form of the DRFPGA [29]. This design is well suited to reliability as opposed to efficiency or security, but the idea can be applied to other areas. In their implementation, CLBs are grouped together into “cells.” Each cell contains four CLBs. Three of the CLBs are used much like any other FPGA to implement specific functions or parts of an overall design. The fourth CLB is used to replace one of the primaries should that CLB fail. The design is relatively rudimentary, but does require significant work to perfect the timing and ensure that routing to/from the CLB is accomplished. Unfortunately, this implementation results in 33% overhead in unused area since the CLB must be maintained as a spare. In addition, once a CLB experiences a failure there is no other spare available. There is a proposed method to arrange the cells in a “super cell” arrangement [29]. This mimics

the cell arrangement in that there are four cells, one of which serves as a backup for another cell in case it should fail. Again, the shortfall comes in that the arrangement will not account for more than one cell failure. Overall the design provides provisions for minimal reconfiguration, but does not provide an acceptable platform for security minded reconfiguration needs as the replacement CLBs/Cells are all precompiled.

*2.3.3 Dynamic Reconfiguration Challenges.* Implementing dynamic reconfiguration is no trivial task. In addition to the difficulties facing any circuit design team, one must also consider the changing functionally and physically relationships between modules. Designers must also accommodate extra space/power requirements for the reconfiguration overhead. Finally, the reconfiguration itself consumes computational effort resulting in lost time and space for the primary functional processes. The penalties associated with reconfiguration circuitry are proportional to the complexity and flexibility of the device. Many researchers have invested time in reducing the overhead of certain aspects only to increase the cost of others. For example, Jean et al. discuss reducing the computational time and space overhead by preprocessing the reconfiguration information with a dedicated Resource Manager (RM) on board the FPGA while the board is in operation [20]. While this may assist in lowering the time necessary to implement the new configuration, it comes with the penalty of increased system complexity and the need for FPGA resources dedicated to the RM. This design does require the new modules to be statically determined prior to the RM placing them. The RM does, however, dynamically allocate space for the new modules. Which reduces the amount of on-board real estate needed for the new processes. The RM is an example of placing more autonomy in the FPGA processes resulting in increased flexibility of the design at the expense of additional computing resources. Placing compilation resources on board (as opposed to using precompiled modules) can further increase the flexibility and reduce space overhead, but processing time and system complexity are sacrificed.

## 2.4 *Polymorphic Reconfiguration*

*2.4.1 Polymorphic Reconfiguration Overview.* The designs discussed thus far represent dynamic configuration, but are not good examples of polymorphic dynamic reconfiguration. They used precompiled blocks that could be substituted for the current functions as needed. Although some designs exhibited granular reconfiguration, they required a precompiled bitstream or bitstream components to overwrite current functions. Polymorphic circuits, on the other hand, provide reconfiguration based on an algorithm rather than implementing static precompiled modules. The computational complexity is greatly increased for polymorphic circuits as compared with DRFPGAs using precompiled modules, but the resulting design is more flexible and better suited to security applications. The primary advantages the polymorphic design has over other DRFPGA designs are granularity and autonomous operation. Since they implement a design algorithm as opposed to strict static module placement, they are capable of implementing a wider variation of designs. They also behave more autonomously since the algorithms typically only require a small set of rules, which leaves the rest to the processor and FPGA attributes.

Most research designs address polymorphic methods as evolutionary in nature. Evolutionary methods work at constructing/reconfiguring the FPGA circuit from the bottom up using a small number of building blocks. This is not consistent with standard engineering top-down design. The general concept is that designs created using small functional circuit blocks combined using a low level algorithm provide an almost natural development process that is largely autonomous. Miller et al. outline some basic defining characteristics of evolutionary methods, a simpler set of rules, building blocks, and an “assemble and test” methodology that differentiate them from more prominent traditional methods of circuit design [23]. Figure 2.6 depicts an abstract representation of the evolutionary and traditional design ideologies. By incorporating evolutionary development facets into the reconfiguration scheme, the overall range of design possibilities increases greatly. Also, by distributing the decision

making to a high number of low level entities instead of consolidating the directives in a single high-level location, the target for adversaries is smaller and more disjointed.

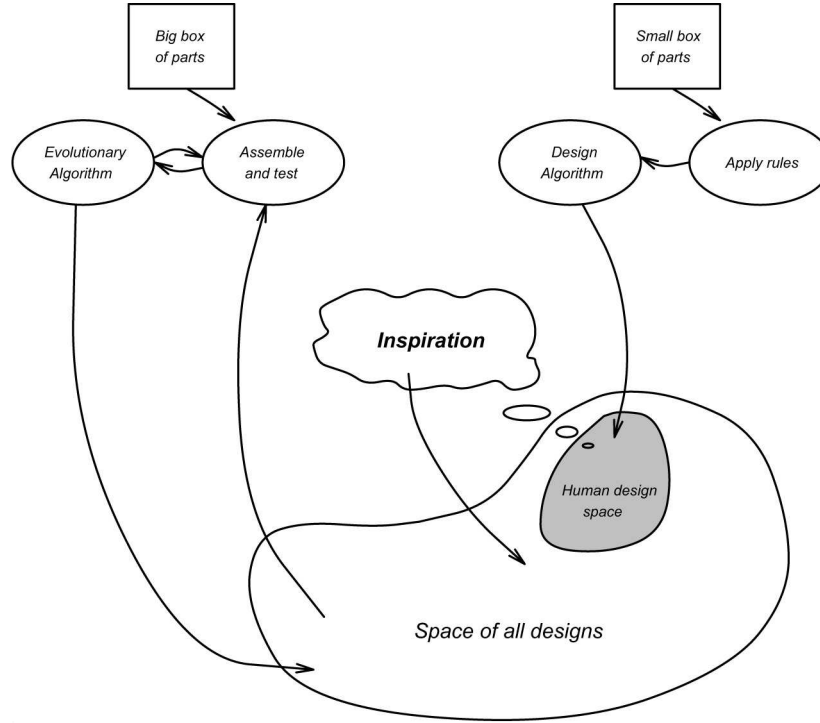


Figure 2.6: An abstract diagram highlighting evolutionary (left side) and traditional (right side) digital circuit design approaches [23].

*2.4.2 Polymorphic Reconfigurable FPGA Designs.* The polymorphic DRF-PGA field contains few examples of fully functioning designs. Most research in the area of polymorphic DRFPGAs focuses on “proof of concept” algorithms or relatively simple implementations of the algorithms. For example, one of the more prolific publishers on polymorphic/evolutionary design, Stoica, discusses a planned architecture that relies on actual transistor level granular reconfiguration for implementation [4]. Reconfiguration is performed at the transistor level where different voltages, temperatures, and other environmental values cause the transistors to behave in varying manners. Gates constructed using these transistors in a Field Programmable Transistor Array (FPTA) will have different functions depending on the voltage, temperature, etc. For example, an “AND/OR gate” will function as an AND gate at lower

temperatures (5° C) or an OR gate at higher temperatures (90° C) [4]. While this implementation appears to have promising uses in the security field, controlling the temperature and voltage in a reliable enough manner to prevent accidental misconfiguration could prove to be an overwhelming task. The following section covers some difficulties in the evolutionary circuit field.

*2.4.3 Polymorphic Reconfiguration Challenges.* While promising in theory, there are still challenges to the implementation of polymorphic dynamic reconfiguration. The practicality of evolutionary design may be in question when it comes to large circuits. Stoica et al. found that the efficiency of large designs was not optimal using the evolutionary model [3]. They theorized that larger circuits could be realized if the complexity of the basic building blocks was increased. Since one of the factors separating the DRFPGAs from polymorphic FPGAs is the very idea of small functional units, designers must be careful they do not cross the boundary into the DRFPGA field, which limits the benefits of implementing the evolutionary design.

Some designs proposed environmental control variables such as temperature or voltage level. External conditions are not always easily controlled and can pose a serious threat to the standard operation of the circuit. Given the critical applications at hand, it is unlikely implementing external controls will provide a format conducive to the job. In addition to random environmental impacts, using external conditions for directing the reconfiguration process provides another attack vector for adversaries.

*2.4.4 Polymorphic Reconfiguration Summary.* As with all technology, FPGAs offer great possibilities, but misuse can seriously hinder any operations relying on the device and result in compromise of sensitive data. As FPGA use becomes more prevalent for proprietary designs, so does the overall community effort to discover ways to view/acquire said designs. One possible way of protecting designs implemented on an FPGA is to reconfigure the design, which creates a mobile target for those seeking to illicitly obtain information from the FPGA. In order to maintain the

operability of FPGA devices, the reconfiguration must be performed while the device is in operation – also known as dynamic reconfiguration.

A substantial amount of research has been performed targeting dynamic reconfiguration for optimization or flexibility of designs, but little has been published on security-minded reconfiguration. Dynamic reconfiguration presents a valid approach to protect designs and, in particular, polymorphic dynamic reconfiguration provides a granular, autonomous method of implementing FPGA design security. The research in this field is limited, but has provided some promising prospects.

It is important for more research to be performed exploring the possibility of implementing security designs using polymorphic reconfiguration. While research has provided promising designs, the shortcomings prevent them from serving as valid solutions for critical applications. More research must be accomplished to investigate a controllable polymorphic design capable of protecting proprietary designs from theft while providing a reliable computing platform.

### III. Research Methodology

The following chapter serves to provide an outline of the implementation and testing phase for research into dynamic reconfiguration in support of FPGA security. Each step of the development and test process is provided for understanding of the actual test results.

#### 3.1 Problem Definition

*3.1.1 Goals and Hypothesis.* The goal of this research effort is to secure FPGA designs against adversarial compromise through the use of polymorphic and dynamic reconfiguration. For simplicity, further discussions in this thesis will refer to all reconfiguration methods as dynamic reconfiguration as opposed to separating the polymorphic and standard dynamic reconfiguration. There are other designs that incorporate dynamically reconfigurable functionality; however, security is not the motivation for reconfiguration. In addition, there are serious shortcomings regarding the granularity of existing reconfiguration methods that limits their effectiveness at reconfiguration for security.

Research supports the belief that a reconfigurable FPGA will provide significant protection against reverse-engineering and/or cloning of the device. Even a relatively simple design from a reconfigurable Xilinx® Virtex4® based FPGA cannot be obtained using the current reverse engineering methods unless the reconfiguration time is longer than the time necessary to discover the design from the original FPGA. The most rudimentary exploration technique, black box testing, is not feasible for most designs found in current systems. For example, to perform an entire black box analysis on a 32-bit adder would take over 1,000 years. Current architectures commonly use even larger designs of 64 or even 128 bits. Therefore, black box techniques are not a primary concern in this research.

The other methods of reverse engineering discussed previously, including passive power analysis, glitching, and bitstream theft, are addressed through the reconfigurable platform and algorithm addressed in this chapter.

*3.1.2 Approach.* The security of an FPGA is improved using a dynamically reconfigurable FPGA design. The proposed design performs reconfiguration “on-the-fly” such that the circuit design implemented on the FPGA is resistant to analysis or reverse engineering. Reconfiguration is accomplished without external interaction in the form of user commands or off-chip computational resources. Everything needed to execute the reconfiguration process is contained within the actual FPGA IC package; thus, the possibility of malicious interaction with the FPGA design is limited. The current research effort does not include the full autonomy, but will support future efforts through autonomous designs requiring modest computational investment on the part of the external controller.

The reconfiguration process must not only retain all the functionality of the original circuit and be executed autonomously, but it must also be completed within a time frame deemed reasonable. This particular time frame is bounded by a value based on the criticality and size of the target circuit. If a user dictates that a circuit must operate with no more than 5 ms of downtime, then this value becomes the upper bound for the reconfiguration time. Although the targeted portion of the circuit will not be operable during the reconfiguration process, the rest of the circuit will be capable of operating with limited impact. The limited impact comes in the form of buffering any inputs/outputs from the targeted portion to other non-targeted portions.

One half of the solution is the development of a hardware platform supporting dynamic reconfiguration. Current FPGA designs are limited in granularity and do not permit the reconfiguration of single CLB s as required for successful execution of the algorithm above. They also lack autonomy at a low level of operation. A custom FPGA architecture has been developed to address these shortcomings and support the reconfiguration process. The custom architecture permits the reconfiguration of any CLB without reloading or reconfiguring any other CLBs through the use of a custom designed serial/parallel communication network. The FPGA is also supplemented

with a similar serial network allowing the system to target any single LUT in the FPGA for reconfiguration via the reconfiguration algorithm.

The other half of the reconfiguration process is the reconfiguration algorithm. The algorithm defines the changes in circuit configuration. For this implementation, the algorithm has two primary avenues for reconfiguration: LUT value inversion and functional replacement. Through completion of either reconfiguration method, the actual composition of the circuit can be altered without impacting the operation. The functional replacement process can also be used to alter the output of the circuit in a manner that is only predictable to processes or modules that have knowledge of the new function output configuration through a key. Use of the key will allow an external process to identify the legitimate and illegitimate output pins for the circuit as well as to adjust the timing to accommodate changes in the timing.

The Xilinx® Virtex4® includes a PowerPC® microprocessor embedded within the FPGA design. The design is structured such that the embedded processor is used to execute the algorithm for reconfiguration using code stored on the FPGA. The efficiency and security of the reconfiguration is improved over an implementation where the microprocessor executes the algorithm from off-chip. Given time constraints, the PowerPC® processes are simulated using a manual process of developing the programming bitstream. The reconfiguration algorithm and hardware are designed to implement as much of the functionality as possible at the CLB or LUT level. This granular autonomy is used to lower the computational overhead of the reconfiguration process as well as limit the transmission of critical data across data busses within the FPGA. By limiting the amount of data transmitted to accomplish the reconfiguration, the odds of an adversary successfully tampering with the reconfiguration process itself are reduced.

After a thorough survey of the DRFPGA field, it became evident that a different approach was needed. The evolutionary algorithm arena provides a manner of reconfiguration that has not been widely implemented in the FPGA fields. Evolution-

ary algorithms rely on fine granularity and autonomous cells to simulate biological processes and remove the constraints of established circuit design procedures. By implementing the granular, autonomous structure, the reconfiguration algorithm becomes truly polymorphic and is limited only by the constraints of the reconfiguration algorithms. However, the evolutionary algorithms currently in use are not well suited to obfuscation of code and typically suffer from inefficient time and power usage. By combining the strengths of granular, autonomous, cellular reconfiguration with selective and predictable (to authorized parties) obfuscation, an ideal secure platform is developed.

### 3.2 Research Boundaries

To evaluate the effectiveness of the DRFPGA, it is imperative that there are clearly defined boundaries between what is within the scope of the research effort. As seen in Figure 3.1, the overall System Under Test (SUT) consists of the FPGA device, software, and the test circuit implemented within the FPGA. This collection of components constitutes the DRFPGA.

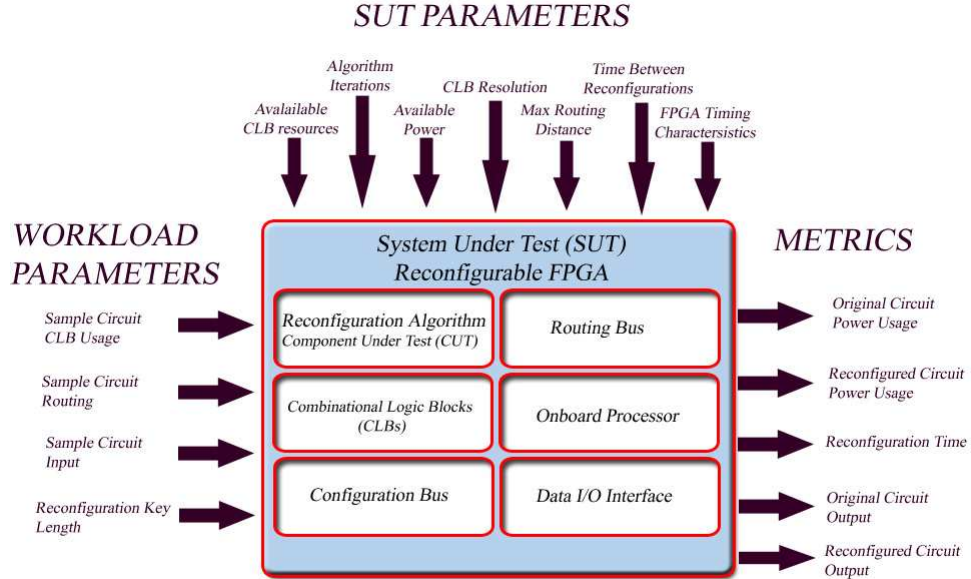


Figure 3.1: Abstract representation of the Reconfigurable FPGA as the SUT.

The Component Under Test (CUT) of the system is the reconfiguration process and support hardware. The DRFPGA is comprised of six primary components: reconfiguration algorithms, CLBs, configuration network, inversion network, routing bus, PowerPC<sup>®</sup> Microprocessor, and the data I/O interface.

**Reconfiguration Algorithm** This is part of the actual CUT. The reconfiguration algorithm is executed manually as of the conclusion of this research effort, but future implementations would be executed autonomously using on-board resources.

**CLBs** The CLBs make up the primary functional components of the DRFPGA. Each CLB houses eight LUTs and devices needed to route the LUT input/output to the routing bus. The circuit functionality is contained within the CLBs.

**Configuration Network** The configuration bus carries the information necessary to configure the CLBs. This is separate from the routing bus, which is used for operational data. The configuration bus also serves to reconfigure the DRFPGA while it is in operation.

**Inversion Network** The inversion network allows the targeted inversion of any LUT within the design. As discussed later in this chapter, the inversion process is carried out in two stages, both of which are implemented functionally at the LUT level. The inversion network carries the command from the initiator (PowerPC<sup>®</sup> core or manually derived instruction) to the selected LUT.

**Routing Bus** The routing bus connects CLB inputs and outputs across the DRFPGA. The bus is not unique to this platform, but is instead based on the Xilinx<sup>®</sup> schematics. This component does not implement the core functionality of the components, but rather the interconnection of CLBs.

**PowerPC<sup>®</sup> Microprocessor** The PowerPC<sup>®</sup> microprocessor is implemented on the DRFPGA and is used to execute the reconfiguration algorithm using the inversion and reconfiguration networks. The PowerPC microprocessor has not been

implemented as part of this research effort; therefore, the instructions delivered through the networks will be compiled manually and programmed into the networks via VHDL test bench stimulus.

**Data I/O Interface** The data I/O interface allows inputs to be passed to the circuit implemented on the FPGA and allows reading of the outputs for the circuit. For the purpose of this research, inputs are provided via VHDL test bench stimulus. The outputs are read directly from the VHDL model during simulation.

Outside of the system boundaries and the scope of research is the support hardware such as the computer used to initially program the FPGA, power supply, and input/output monitoring equipment. Since a VHDL model is being used for the test runs, the actual FPGA hardware and factors affecting its operation (e.g., temperature, voltage) are not contained within the scope of this research. In addition to the components listed within the SUT, there are also parameters and metrics, which are discussed in their respective sections.

This research effort will not study the actual attributes of the physical implementation of the design except as it applies to the functionality of the circuit, reconfiguration algorithm, and hardware support for dynamic reconfiguration. The VHDL model includes the custom architecture design permitting individual CLB addressing, but will not include the actual PowerPC<sup>®</sup> processor. The steps taken by the PowerPC<sup>®</sup> are simulated using a manual process to develop LUT and reconfiguration instructions while operating under the constraints of the algorithm. The design of the DRFPGA is based on the Virtex4<sup>®</sup> FPGA with added hardware supporting the individual CLB and LUT addressing methods. For this research effort, the reconfiguration algorithm is limited to operating on relatively simple circuits. The goal does not include stress testing the design using large data sets. Typical test circuits include simple designs such as adders, counters and comparator circuits (specifics are discussed further in the workload section).

### 3.3 System Services

The purpose of the SUT is to implement an original user provided circuit and provide a secure, reconfigured circuit that is functionally identical to the original, except in the case where spatial output obfuscation is desired. With each reconfiguration operation, the reconfigured portion of the circuit returns to normal operation within the reconfiguration time overhead (dependant on the circuit implemented). Inputs are provided to the DRFPGA, processed by its programmed circuit, and provided as outputs for use or analysis. After reconfiguration, the circuit will either operate as expected given the reconfiguration and the decoy circuit output routing or the output will not match. Differing output represents a failure in the reconfiguration process unless the output differentiation is planned. Failure of the SUT can occur either due to an incomplete reconfiguration execution (e.g. the algorithm is interrupted) or the reconfiguration completes, but the result is not functionally equivalent to the original circuit. Given the two possible variables (reconfiguration completion and correct reconfiguration) and the two values per variable (failure or success), there are a total of four potential outcomes of the process as dictated in Table 3.1. Although the possibility exists for the reconfiguration to fail and the outputs to match the expected outputs, this does not represent a successful SUT outcome.

Table 3.1: Potential outcomes of SUT

Output Correct	Reconfiguration Succeeds	SUT Success
No	No	No
No	Yes	No
Yes	No	No
Yes	Yes	Yes

### 3.4 Workload

Given the varied uses of a design and the limited functionality needed at this time, the workload selected is simple in design and operation as compared to the

final expected circuits since the goal is to verify the correctness of the reconfiguration algorithm and the feasibility of the implementation. The workload consists of binary counters, ripple carry adders (with carry in), and comparators. Specifically, 4-bit and 8-bit versions of the adder and binary counter and an 8-bit variant of the comparator circuit provide a total of five sample circuits. The 4-bit variants are used to implement the 8-bit circuits and are not independently evaluated because they are included in the 8-bit test vectors.

In addition to the circuit inputs supplied to the SUT, there is also the requirement for a *reconfiguration key*. The *reconfiguration key* determines the pseudo-random target outputs for the decoy circuits. The key is currently implemented as a seed for the Linear Feedback Shift Register (LFSR) selecting the reconfiguration target. Future implementations will also designate valid and invalid outputs. While the key is provided as part of the workload for the purpose of evaluation, it will be stored on the chip for any actual fielding of the DRFPGA.

### 3.5 *Performance Metrics*

The SUT is evaluated based on measured performance metrics. The four selected metrics are: power usage, time to reconfigure, area overhead for reconfiguration logic, and the accuracy of the reconfigured circuit output.

**Power Usage** The power needed to perform reconfigurations is compared between the DRFPGA and FPGA. The measurement will capture the power consumed from the first issuance of the reconfiguration command until the system is ready to accept its first input signal for the reconfigured circuit. Evaluating this metric will provide a method of determining whether the implementation of a reconfigurable platform yields power benefits when compared to the FPGA.

**Time to reconfigure** After the system performs a task using a sample circuit (e.g., an adder) and input, a command is issued to reconfigure the circuit. The time

from the first issuance of the reconfiguration command until the system is ready to accept its first input signal for the reconfigured circuit is called the *time to reconfigure*. Reconfiguration time impacts operation since the DRFPGA CLB s targeted for reconfiguration are not available during the reconfiguration. For example, if an operation normally takes 100 clock cycles to complete and the reconfiguration takes five clock cycles, then the operation running on the DRFPGA will take 5% more time to execute the operation.

**Area overhead for reconfiguration logic** The area for an FPGA implementation with reconfiguration functionality is calculated and compared to a functionally similar FPGA without the reconfiguration hardware. This metric serves to highlight the feasibility of the system in limited space/weight applications such as space vehicles or portable systems.

**Accuracy of reconfigured circuit output** The non-reconfigured circuit output is compared to the output of the reconfigured circuit given the same input values. Given the relatively simple sample circuits chosen, exhaustive tests can verify that the input is 100% accurate. Accurate circuit output is vital to any operation.

### 3.6 Parameters

*3.6.1 SUT Parameters.* The SUT parameters are variables impacting the outcome of the dynamic reconfiguration process. The SUT parameters are: available resources, algorithm iterations, CLB/LUT resolution, maximum routing distance, time between reconfigurations, and FPGA timing characteristics.

**Available resources** This parameter describes the size of the FPGA in CLBs. The number of CLBs limits the maximum size sample circuit that can be implemented.

**Algorithm iterations** The reconfiguration algorithm is executed iteratively to strengthen the security for the resulting reconfigured circuit. For the LUT inversion method,

continual iterations throughout the circuit operation will guarantee a bitstream that cannot be stolen and implemented on another DRFPGA.

**CLB/LUT resolution** The reconfiguration resolution refers to the smallest component that can be targeted for reconfiguration. The module size targeted for reconfiguration varies from a single LUT entry to the entire DRFPGA. For the cases considered in this research, the resolution varies either at the LUT level(LUT inversion algorithm) or CLB level (functional replacement).

**Maximum routing distance** Routing signals between CLBs takes time and consumes power. To maintain correct operation, an upper limit must be imposed on routing lengths between CLBs. This value may be altered for specific applications. For example, low power applications would require shorter routing lengths at the cost of less flexible reconfigurations. If security and speed of reconfiguration becomes the goal, the routing lengths can be increased providing more options when placing CLB components. For the purpose of this research, the routing distance is across four CLBs. This allows the measurement and analysis to be focused on the reconfiguration algorithms and hardware aspects as they are critical to the development of a DRFPGA platform.

**Time between reconfigurations** The time between reconfigurations is varied to increase security or to decrease latency in operations. This parameter is not to be confused with algorithm iterations. For example, if the algorithm iteration parameter is five, then five iterations of the algorithm are ran successively and separated by the time between reconfigurations. After waiting for the time between reconfigurations, five iterative algorithm executions are performed again. Increasing the time between reconfigurations will allow more operational processing to occur at the cost of lower security while decreasing the time will shorten the analysis opportunities for adversaries seeking to study the circuit but negatively impact operation processing efficiency.

**FPGA timing characteristics** Timing characteristics encompass the delay parameters dictating a circuit’s operating speed. The fundamental defining attribute for timing will be the delay between input and output on combination circuits (adder and comparator) or the maximum delay between storage elements in sequential circuits (counter).

### *3.6.2 Workload parameters.*

**Sample Circuit CLB usage** This is the number of CLBs that are needed to implement the sample circuit. CLB usage will affect the power needed for the circuit as well as the time to reconfigure. A larger circuit generally uses more power and takes more time to reconfigure.

**Sample Circuit Routing** The connection routing between CLBs for the sample circuit affects the power usage and reconfiguration time.

**Sample Circuit Input** Varying the sample circuit input is necessary to verify that the reconfigured circuit correctly operates. Since the 4-bit sample circuits used are relatively simple, full factorial input vectors are used to test for correct operation. The 8-bit variants use a full factorial test for each 4-bit sub-circuit. The 8-bit circuits have been designed in a modular manner that allows the upper 4-bit circuit to be tested independently of the first four bits as long as the carry bit from the lower four to the upper four is included in the upper 4-bit module’s full factorial test.

**Reconfiguration Key Length** The reconfiguration key is used to initialize the LFSR selecting the target for reconfiguration. The length of the key affects the security and efficiency of the reconfiguration effort. A larger key results in a more secure circuit by providing more options for the random node placement, but requires more storage and computation.

### 3.7 Factors

Factors provide a way to alter the experiment and find ways to improve the process or identify potential problems. The following factors are taken from the list of parameters affecting the SUT and experiments:

**Sample Circuit** The circuit makeup affects the time to reconfigure different circuits and can have a significant effect on the process results. The DRFPGA is programmed with three different types of circuits: binary counters, ripple-carry adders, and comparators. Each circuit type is implemented in 4-bit and 8-bit variations except for the comparator. The four basic designs represent common components of larger systems. By varying the complexity of each circuit type through the adjustment of the bit width, varying degrees of CLB usage are evaluated while still retaining an achievable goal of total factorial testing. The next common input size, 16 bits, will increase the number of test cases for full factorial testing by a factor of over 65,000 times for the adder and comparator circuits. Diagrams of sample circuits expressed as CLB block diagrams are located in Appendix D.

**Sample Circuit Input** Each input circuit is evaluated using every input combination available. For an 8-bit circuit (the most complex test configuration), there are 131,072 different input combinations. Given the time required to implement each of the test vectors, the testing method uses overlap between subsequent 4-bit modules to reduce the number of vectors to 64. Since the counter does not accept user input aside from the enable and clock signals, the vectors for its test are relatively simple. The counter is simply allowed to progress from the initial state (all 0's) to the final state (all 1's). This process takes 256 clock cycles. For all test circuits, the output for every input test is compared with a known good output to verify accurate operation.

**Time between Reconfigurations** The reconfigurations are not being run continuously. Otherwise, for the relatively simple designs implemented as test circuits,

the actual circuit implemented on the DRFPGA would never perform its function. The time between reconfigurations dictates how much time the circuit will have to run. Shortening the time between reconfigurations decreases the target time for adversaries to inspect the DRFPGA hardware/operations as a static system; however, this also decreases the efficiency of the sample circuit implemented on the DRFPGA. On the other hand, increasing the time between operations provides more time for the sample circuit operation and a longer static target for adversarial inspection and analysis. The time is designated based on the time needed to successfully map the circuit using black box reverse engineering attacks.

### **3.8 Evaluation Technique**

The DRFPGA is evaluated using simulation software on the VHDL model. The software is relatively straight forward and will not include testing of environmental variables on the operation of the DRFPGA. The algorithm is developed and tested for the target FPGA ( Xilinx® Virtex4® ) and will not be modeled for implementation on other hardware. Each of the three primary circuit variations listed in the factors section is programmed on the DRFPGA in 4-bit and 8-bit variants, except for the comparator, which is implemented only in an 8-bit variant. For each circuit programmed (regardless of whether it is on the DRFPGA or the FPGA), the following steps are performed:

**Program** Compile the VHDL model in the simulation software.

**Operation** Circuit simulation using the test input combinations in a sequential manner. For example, on the 4-bit adder the first run would be 0000 + 0000, for input A + input B. This input can be viewed as input 00000000, where the first four bits are input A, the next four input B. The next input is 00000001. In this manner, the input is varied from all 0's to all 1's.

**Power Measurement** Simulation software used to measure the maximum power usage.

**Execution Time** Execution time is measured for the total run of all inputs.

**Output** Output values for each input are compared to the output of a known good circuit. Any discrepancies result in immediate flagging as a failed run and no further input vectors are executed.

**FPGA CLB Count** Total area (in CLBs) is documented. The CLB count is provided by counting the CLBs used in the VHDL model.

Each of the results for a specific circuit will be compared between the initial circuit and the reconfigured circuit. This will provide values as discussed in the metrics section.

### ***3.9 Experimental Design***

The experimental design mimics a full factorial test. To test the 8-bit circuits, the circuit is first broken down into two 4-bit circuits with a single signal carrying from one 4-bit section to the next. Every combination of the first 4-bit circuit is tested, followed by the second circuit tested in the same manner, except that the carry signal is simulated to provide the second circuit with every possible signal it could receive. Every combination of factor values is explored and the results compared. This provides the means to determine the implications of implementing the DRFPGA for the selected circuits which, since the circuits selected represent common sub-components of most real-world circuits, accurately reflect the DRFPGA performance in fielded applications. The total number of experimental inputs applied per full factorial run is listed in Table 3.9.

The total estimated time to run each full factorial test on the simulation is based on an average time of one second per test (measured mean). Therefore, the total test time given no failures is 37.55 hours.

Table 3.2: Experimental input counts.

Test Counts per System (DRFPGA/FPGA)			
	Original	Reconfigured	Total
8-Bit Adder	1,536	1,536	3,072
8-Bit Counter	512	512	1,024
8-Bit Comparator	65,536	65,536	131,072
Totals	67,584	67,584	135,168

Since the test is conducted as a simulation of a VHDL model, multiple runs will not be necessary for any given configuration of the SUT. No random variables will be added to the test runs so the results will be identical from each execution of the full factorial test to the next unless factors are adjusted. If a failure is found, then that test must be reaccomplished after fixing existing issues.

### 3.10 Methodology Summary

This section serves to document the design and experimentation of the DRFPGA. The DRFPGA is proposed as a solution for a secure FPGA platform protecting circuit designs from exploitation, theft, and unauthorized duplication. The key distinguishing factors for a DRFPGA are the reconfiguration algorithm and the hardware network supporting granular reconfiguration. It is important to consider the parameters affecting the DRFPGA performance before initializing the experimentation process. The reconfiguration algorithms and supporting hardware networks are the primary foci of experimentation. Other key parameters are the DRFPGA hardware attributes (e.g. CLB resources and timing characteristics), actual circuits implemented on the DRFPGA, and the time between reconfigurations. The circuits, reconfiguration targets, and reconfiguration methods are varied through the experimental design as *factors*. Through manipulation of the *factors*, differing results are found that can be attributed to the adjusted value. Important results include the time necessary to perform a reconfiguration, the power overhead involved in providing a reconfigurable system, and the hardware resource (in CLBs) of implementing

a DRFPGA as compared to the FPGA. The manipulation of the factors results in a total of 4,286,976 combined test cases for the DRFPGA and FPGA devices.

Overall, implementing this experimental model adequately addresses the need for evaluation of the DRFPGA device using simulated runs of a VHDL model. Through analysis of the results, shortfalls are identified and further research directions provided.

### ***3.11 Custom FPGA Design***

The functional requirements of the reconfigurable platform mandate a custom FPGA design as no design currently exists that supports the granularity of reconfiguration needed. The custom design is based on the Xilinx® Virtex4® architecture, but includes substantially more capabilities in the retiming and reconfiguration arena. The framework was constructed by Jason Paul in support of his work on retiming FPGAs for security [37]. The FPGA was created in two primary phases: one to model the CLB and data routing functions and the other to implement a programming router network. The CLB hardware foundation was created by Paul in 2007 as a hardware platform supporting retiming.

The original design consisted of a 4x4 array of CLBs arranged in an FPGA. For this research effort the CLB count remains the same, although the overall size (in area and components) is larger due to the addition of the reconfiguration hardware. In order to permit precise dynamic reconfiguration, a programming network has been created and added to the FPGA design to allow the real-time programming of any single CLB or the entire FPGA.

The FPGA model is constructed in a hierarchical manner allowing the abstraction of function and data requirements for the modules within the design. The standard VHDL programming language was selected for the FPGA model based on its acceptance as an industry standard. Each component has clearly defined functional requirements as outlined below.

**3.11.1 CLB.** The CLBs are used to implement discrete functions within the FPGA. Multiple CLBs are connected together to create the overall design. For example, a 4-bit adder can be constructed of four 1-bit adders connected together. Each 1-bit adder can be implemented in a single CLB. The four CLBs are interconnected using the data bus to implement the complete 4-bit adder design. The CLB structure is based on the CLBs found in the Xilinx® Virtex4® line of FPGAs. Each individual CLB consists of eight total LUTs capable of implementing custom functions as well as 32 bits of RAM/ROM storage (configurable as a 32x1-bit or a 16x2-bit memory element), Multiplexors (MUXs), Flip Flops (FFs), and Latches. For ease of programming and comprehension, the components within the CLB are divided into four Slices (two M-Slices and two L-Slices). The schematic diagram of a single CLB is shown Figure 3.2. The CLBs are interconnected through the use of a switch matrix connecting output and input lines to selected buses in the fabric. The fabric represents the data network responsible for passing data between different CLB s. It is important to note the data network is completely separate from the configuration and LUT inversion networks.

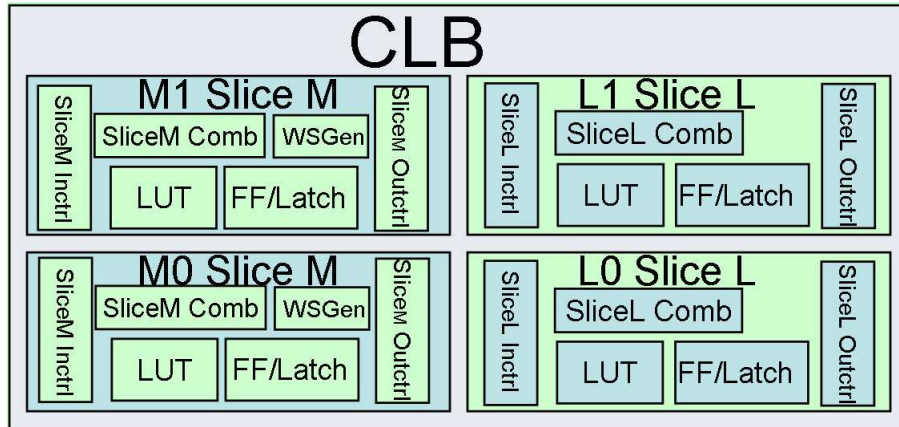


Figure 3.2: VHDL model of a CLB. Each labeled block represents a VHDL sub-module at the bottom of the hierarchy (i.e. not composed of other sub-modules). The data and programming networks are omitted for clarity. A schematic view of the CLB is depicted in Appendix A.

**L-Slice** L-Slices contain two 4-input, 16-entry LUTs and the switching hardware necessary to direct the input and output of the slices. They also contain two components that can be configured as FF's or Latches, which are used to implement sequential devices.

**M-Slice** The M-Slice contains all of the functionality of the L-Slice, but adds memory functions and the capability to serve as a shift register.

**Input/Output Control Module** The Input/Output control modules dictate where the input and output lines are connected for the slices. This should not be confused with the switch matrix or data bus routers as the Control Modules only direct the information flow within the CLB.

**LUT** The LUT modules serve to implement the fundamental functions. The reconfiguration algorithm is functionally contained within the LUT hardware although it is not performed unless the instruction is delivered through the LUT inversion network.

*3.11.2 Control Registers.* The control registers are used to configure the FPGA for operation. Each CLB is connected to a single control register. LUT configuration, data contents, and data directions are all dictated at the configuration time and based on the contents of the control registers. To program the circuit using a serial JTAG method, the control registers include a shift-in and shift-out bit that is used to implement a chain of registers. In addition, the Programming Network is connected to the control registers allowing dynamic reconfiguration.

*3.11.3 FPGA Data Network.* The data network is composed of the data bus (or fabric) and routers. The routers serve to connect certain lines in the data bus together creating a data path between CLBs. The data routers are configured during the initial programming stage. The routers serve as the equivalent of a multi selectable input MUX. Any input to the router can be routed to any output from the

router. Multiple outputs can be fed from the same input, but to avoid conflicting signals only one input can drive any output.

*3.11.4 CLB Programming Network.* The CLB Programming Network is composed of routers that allow the targeting of individual CLBs for reconfiguration. It is similar to the data network because it is composed of routers and data lines, but the actual data line is only one bit wide. Configuration data is streamed to the target CLB in a serial manner. While this increases the overall configuration time over a parallel load, it significantly reduces the space/power requirements. The programming network is further dissected into three primary subcomponents: chain, channel router, and sub router.

**Chain** There are a total of 8 chains feeding the configuration data to the CLBs. Each chain is responsible for 32 CLBs.

**Channel Router** Each chain has a channel router that directs programming bits to the components within the chain if the targeted CLB is one of the 32 for which the chain is responsible. The channel router is also responsible for resetting the chain components and preparing the sub routers to receive the address for the target CLB.

**Sub Router** Every chain has a channel router and eight sub routers. The sub routers are controlled by the channel router and do not initiate address collection or start processing the bitstream until told to do so by the channel router. Each sub router is responsible for four CLBs. Once the target CLB has been identified via an 8-bit address code, the responsible sub router starts the transfer of data bits to the CLB control register. While not an actual programming network component, the control register plays an important part as the final destination for the programming bits. Once the control register has been programmed with the necessary bits, the actual CLB configuration takes place.

To program the CLB array (or any single CLB), the programmer must first send the address of the target CLB. After sending the address through the input line of the programming network, the bitstream is transmitted serially to the target CLB through the network. The CLB can be programmed during runtime, provided the CLB itself is not in operation or the input sequences are not critical to the overall circuit function.

*3.11.5 LUT Inversion Network.* The LUT inversion network is similar in structure to the FPGA Programming Network. The primary difference is in the addition of MUXs and registers contained within the CLBs to decode the instructions and deliver them to the appropriate LUT within the CLB. Since the network is serial and the LUT and MUX instructions are needed in parallel, the registers store the data and provide them to the LUT and MUX in a 6-bit wide parallel configuration. The chains, channel routers, and sub-routers all function the same as the components from the FPGA Programming Network.

*3.11.6 DRFPGA Programming.* Standard FPGA designs use a serial JTAG interface for programming. The serial data line is composed of the control registers connected in a chain through the FPGA. By using the control registers as shift registers, the data is clocked into the FPGA as a single bitstream. The custom design proposed in this research contains the serial programming functionality in addition to the CLB addressable network previously discussed. The JTAG programming method results are compared to the CLB addressable network. Of interest in this comparison are the approximate power usage, area required, and time to reconfigure. The most critical attribute is the time to reconfigure.

### ***3.12 Reconfiguration Algorithm***

The reconfiguration algorithm dictates the process of altering the circuit while maintaining correct functionality or, in some cases, creating a deterministic method of obfuscating the output. The process is performed in two different methods: inversion

and functional replacement. The end product of the inversion method results in a circuit functionally equivalent to the original circuit. The functional replacement method may be used to create a functionally equivalent circuit or can provide a way to redirect inputs/outputs or even change the timing and design of the circuit completely. In this manner, it is more flexible than the inversion technique, but requires more time due to reprogramming the entire CLB and is not as autonomous (i.e. it requires significant work to create the functional replacement modules). Each step is specialized to operate on FPGA architectures as opposed to Register Transfer Logic (RTL) schematics. The benefits to this approach are two-fold: less overhead for processing and increased security.

One of the concerns regarding reconfigurable FPGAs is the added overhead of the reconfiguration process. The proposed design eliminates a substantial portion of the overhead involved in reconfiguring a circuit because it does not require the translation from RTL or designer level functional diagrams to the architecture of the FPGA. By relating the attributes of the reconfiguration process directly to FPGA modules and components (i.e. LUTs, FFs, or CLBs) the translation between circuit schematics to FPGA implementations is omitted. Most reconfigurable architectures employ a partitioning step similar to Alpert's use of the Fiduccia-Mattheyses partitioning algorithm [6]. This partitioning consumes a great deal of system resources and must be performed after every reconfiguration unless the designer maintains a complete graph representation of the circuit on the FPGA device. Either method sacrifices limited computational resources and adds complexity to the overall process. By avoiding the partitioning process altogether, the proposed reconfiguration method eliminates the performance and time penalties associated with partitioning.

Another concern regarding partitioning of the circuit and/or storage of non-native representations of the circuit is related to the security of the FPGA architecture. Since there has not been extensive research into the use of reconfiguration for the security, this problem has not been well addressed. Recall, adversaries may be able to view the contents of certain memory cells through the use of specialized imag-

ing equipment. If the schematic of the circuit is stored anywhere on the chip, there is potential the adversary may acquire a complete schematic of the circuit. Even if the circuit is not stored permanently at any given time, there are small portions of the circuit stored for the purpose of reconfiguration and placement that are vulnerable. This means that the adversary may be able to acquire small portions of the circuit and, over time, assemble these into a complete overall schematic. These potential pitfalls are avoided altogether through the implementation of a native, organic reconfiguration algorithm.

*3.12.1 LUT Inversion Algorithm.* The inversion process is similar to the bubble pushing concept used in gate level circuit implementations and demonstrated in Figure 3.3. Although pushing bubbles on the gate level circuit description is complicated (given the partitioning and translation from the physical implementation on the actual FPGA), it can be accomplished by focusing on altering the bubble pushing idea to pertain to the LUTs. For example, the function  $F = (A + B) \cdot (C \cdot D)$  can be implemented gate-wise as seen in Figure 3.3 Schematic 1.

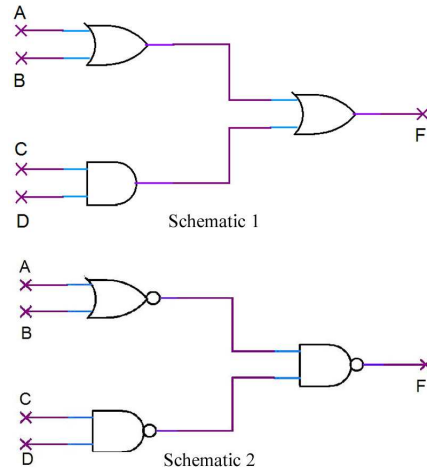


Figure 3.3: Simple demonstration of the bubble pushing concept on a gate level circuit schematic. The circuits in Schematic 1 and Schematic 2 both behave identically in spite of their different structures.

The circuit is reconfigured by pushing the bubbles, creating inverters on the outputs of gates and on the inputs on the following gates, and converting them to

equivalent gates. The same circuit after pushing the bubbles is shown in Figure 3.3 Schematic 2. While the circuits are schematically different (reconfigured), they both will produce identical outputs for any given input combination.

The same concept can be applied to LUTs within a CLB or FPGA. The LUTs are treated like gates, but with a specialized set of rules. LUTs can be thought of as a RAM module storing bits in certain addresses accessed through the inputs. The first step in pushing bubbles in the LUT process involves inverting the values in the LUT, which equates to inverting the output on a gate. Then, select entries in the LUTs connected to the output of the inverted LUT are exchanged. The results of a LUT inversion process are shown in Figure 3.4. As in the gate-wise implementation, Schematic 1 represents the circuit before reconfiguration and Schematic 2 is a functionally identical circuit post-reconfiguration.

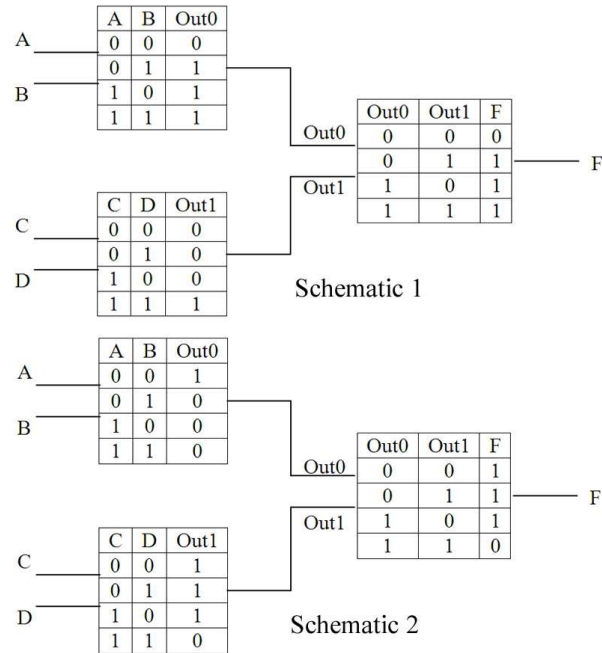


Figure 3.4: Simple demonstration of the pushing bubbles concept as applied to the LUT schematic. The circuits in Schematic 1 and Schematic 2 both behave identically in spite of their different LUT contents.

The manual algorithm for inverting LUTs is more complicated than the basic gate method, but the hardware implementation is significantly easier with the au-

onomous LUTs created for this research and the algorithm’s compatibility with the LUT hardware attributes. Any gate level circuit can be translated to the LUT realm as long as no single gate has more inputs than the LUTs. For example, if a schematic has only 2-input gates, then the circuit can be translated to a LUT circuit completely composed of the 2-input, 4-entry LUTs found in Figure 3.4. In the example, all of the LUT contents were inverted. This is not always the case and only single LUTs are inverted at any given time for the actual reconfiguration. This takes advantage of the flexibility of the LUT as compared to gates. If only one of the gates feeding the output OR gate in Figure 3.3 Schematic 1 were inverted, then the OR gate itself could not be translated to an equivalent gate. LUTs, however, provide additional rules and flexibility that permit single LUTs to be inverted. The operation is governed by a unique set of rules developed for this research effort. Although other sources have performed LUT inversion [39], there are no step-by-step algorithms available for performing single LUT inversions followed by subsequent connected LUT reconfigurations to account for the inverted values. A LUT specific equivalent of the bubble pushing algorithm applied to the circuit shown in Figure 3.3 is depicted in Figure 3.4. A more detailed description of the process is demonstrated in Appendix C.

To further strengthen the obfuscation of the circuit, stages for different reconfigurations can be overlapped. For example, while one LUT system (one LUT output feeding multiple LUT inputs) is in stage one of the inversion process another unrelated LUT is in stage two. Given the current components, this configuration would require either two parallel inversion networks or the FPGA would be segmented into two different regions administered by separate inversion networks.

By performing the inversion steps in an overlapping manner, the circuit bitstream in its entirety would never be correct. The actual bitstream of the LUT system is incorrect because of the initial inversion of the first stage. It is not until the completion of stage two that correct operation is maintained. With this method of reconfiguration, any static adversarial capture of the entire FPGA bitstream would result in an incorrectly functioning circuit.

*3.12.2 Functional Replacement Algorithm.* The DRFPGA has, in addition to the LUT inversion method of obfuscation, the capability to perform functional replacement. This method of obfuscation is similar to other functional unit reconfiguration such as Lodi et. al.'s VLIW FPGA [7]. The primary difference is replacement modules, which are constructed in real-time from a library of sub-modules stored on the chip. While the hardware for reprogramming modules is completely functional as of this research effort, the actual library of sub modules and the module identification method specifics are an area of future study. This research effort serves to outline the fundamental process and provide a roadmap for implementation.

In future research, the functional replacement algorithm is performed by the on-board PowerPC processor executing a semi-intelligent method of module identification, decomposition, and recomposition. The process can be used to create functionally equivalent circuits similar to the bubble pushing algorithm described for RTL circuits, or can actually be used to obfuscate the circuit via output/input relocation and retiming. The following sections outline the fundamental process and provide examples of how specific circuits can be replaced with functional equivalent circuits and obfuscated circuits.

The functional replacement algorithm is a top-level outline of a method for replacing functional systems with new systems. The granularity of the system may be as small as a single CLB or may contain the entire FPGA. Likely the upper limit for the granularity will be the computational overhead and time penalty imposed through the reconfiguration. There are two primary categories of replacement: functionally identical replacement and obfuscated replacement.

**Functionally Identical Replacement** The system selected for replacement is replaced with a system that will produce the exact same output for any input as the original system. In addition, the timing of the replacement system must be the same. Although the overall timing is identical, timing within the system may be adjusted. For example, assume that the original system can be decom-

posed into sub-module-A and sub-module-B with the output of sub-module-A feeding the input of sub-module-B. Sub-module-A may be sequential and require one clock cycle to latch the data onto the output while sub-module-B is combinational and provides the output within a clock cycle. A functionally identical system could consist of two modules in a similar configuration, but with a combinational sub module feeding a sequential sub module. Although the timing within the system has been altered, the change is completely transparent from the system’s external inputs/outputs; thus, the systems are functionally identical when considered as a black box system.

**Obfuscated Replacement** Circuit details can be further secured using temporal and spatial obfuscation. Temporal obfuscation is achieved using retiming as in Paul’s research effort [37]. Using temporal obfuscation, the output timing is altered by changing the overall clock cycles required to process the input and provide an accurate output. For example, if an adder is implemented as a combinational circuit then the output for any given input is available within one clock cycle. By inserting delays via FFs, the output is shifted temporally by at least one clock cycle. Any adversary attempting to analyze the results would not know when the correct output is available without a comprehensive understanding of the reconfiguration process. Spatial obfuscation is similar, but the output is placed on different physical pins as opposed to being shifted in time. This serves the same purpose as an adversary would not be aware of which pins contain the appropriate output (or what order the pins should be read in) without a comprehensive understanding and insider information.

In order to explain the abstraction of functional replacement, certain attributes of the overarching design must be identified and defined. The following attributes will be referred to throughout the process description:

**Target System** For the purpose of describing the abstract replacement process, the target system refers to the collective group of components that will be replaced.

This includes all of the components necessary to implement the function selected for replacement. For example, in the case of FPGA system replacement, the system would include the CLBs, routers, data lines, and input output lines.

**Sub-module** A sub-module is a system identified by its input/output relationship. Examples of sub-modules applicable to the functional replacement algorithm may be adders, multipliers, or even simple gates. The size of a sub-module depends on its function and can vary from granular (i.e. an AND gate) to coarse (i.e. a binary divider) depending on the function. The key is that multiple sub-modules may be schematically different, but still implement the same function. This idea is critical functional replacement.

**Library** The library consists of all available, precompiled sub-modules. Each sub-module is identified by its input/output width, timing, size, and function. For example, an XOR gate may be identified as having a 4-bit input/1-bit output, combinational (timing of zero clock cycles), 1-LUT (or CLB depending on chosen granularity) configuration. Functionally, this sub-module could be replaced with any 4-bit input/1-bit output, combinational XOR gate regardless of size. It could also be replaced with a sequential (as opposed to combinational) configuration to effectively retime the circuit; however, this would have to be taken into account at the external output. The library contains multiple instances of functionally equivalent sub-modules for replacement as well as sub-modules that are slightly different in spacial or timing characteristics such that the replacement can be functionally equivalent or may provide a temporal or spacial offset for the purpose of obfuscation or optimization.

Both categories of replacement follow the same algorithm structure. The algorithm is described in an abstract sense in the following stages: system boundary identification, system decomposition, sub-module replacement, system recomposition, reconfiguration. The definitions of each step are described below:

**System Boundary Identification** Initially, the system chosen for replacement must be defined. The granularity of the selection can be defined as a single LUT or entire DRFPGA. The size of the system will affect the time necessary to accomplish the replacement itself. In identifying the system, the inputs and outputs must be listed. If the replacement is functionally identical, then the inputs and outputs of the final reconfiguration must match identically to the current configuration. For obfuscated replacement, the inputs and outputs may not match functionally, but each external connection must be connected to an internal function to prevent a complete failure of the entire device.

**System Decomposition** The system, as identified in the first stage of the replacement process, must be broken down into sub-modules that can be compared to the library modules. The size and configuration of the modules are determined by the library contents.

**Sub-Module Replacement** The library contains multiple instances of each sub-module that are functionally equivalent. In addition, there may be spatially or temporally obfuscated circuit representations. The equivalent sub-modules will provide identical output for the entire input space, but will accomplish the execution in a different manner.

**System Recomposition** After selecting the sub-modules for replacement, the targeted system must be reassembled to match the overall external input and output configuration of the original circuit.

**System Reconfiguration** Once the targeted system has been reassembled, the entire system is stored in the FPGA. The reconfiguration takes place one CLB at a time. To minimize the impact, the reconfiguration should start with the lowest numbered CLBs (as arranged in a leveled manner).

*3.12.3 Obfuscation Mask.* Obfuscated replacement differs from functional only replacement primarily in the addition of a mask designating the correct output locations (spatial obfuscation) or timing (temporal obfuscation). The mask must be

known to the program receiving the reconfigured circuit outputs. The specifics of mask design are not within the scope of this research, although the functional replacement example does demonstrate spatial obfuscation for the purpose of security. The mask itself is a static representation of the reconfiguration algorithm starting point and is dependant on the reconfiguration key. The DRFPGA and external interface systems must be synchronized such that legitimate systems interacting with the DRFPGA can predict the current input and output locations. The mask does not serve to simply block the incoming or outgoing signals, but redirects data targets (input side of the DRFPGA) or sources (output side of the DRFPGA) for proper operation.

The obfuscation mask was not developed for the examples in this research because of the limited output lines. The mask is an essential part of reconfiguration for more complicated designs.

### ***3.13 Summary***

The DRFPGA SUT was tested in accordance with the specified workload parameters, factors, and metrics. Both the functional replacement and LUT inversion methods of reconfiguration are evaluated for functional accuracy and efficiency parameters. Three 8-bit test circuits were constructed for the DRFPGA: ripple carry adder, counter, and comparator.

## IV. Test Process and Results

This chapter describes the details of the test implementations and results. The test details are provided for both the inversion and functional replacement techniques. After reconfiguring the circuit, the altered circuit is tested for correct operation.

### 4.1 Test Circuits

*4.1.1 Test Circuit Overview.* A variety of test circuits are used to demonstrate the effectiveness of reconfiguration and to measure the penalty in performance and power usage associated with the reconfiguration process. Given the intense manual cost of implementing the individual circuits, the test circuits are limited to relatively simple devices that might be used in the construction of larger, more complex systems. Each test circuit has been implemented in 4-bit and 8-bit variations except for the comparator circuit, which is implemented in an 8-bit variant only. The functionality for the circuits is contained within the CLB LUTs. Generalized LUT content for each circuit can be found in Appendix C. An overview of the test circuits is provided below:

**Adder** The adder circuit accepts two binary numbers as input and provides two output values: the sum value and the carry out value. An 8-bit adder was developed for this research using two 4-bit adders.

**Counter** The counter circuit accepts an enable input and a clock input and provides a binary number as output. As long as the enable input is 1, the counter will increment the binary number output. Once the binary output reaches the maximum value possible (all bits = 1) it will restart from the zero state (all bits = 0).

**Comparator** The comparator circuit accepts two binary numbers as input and provides a single bit output. The output is 1 if the two input numbers are identical else it is 0.

In addition to implementing the functional attributes of the test circuits in LUTs within the FPGA, the routing between different functions is created to maintain proper operation. The functional LUT schematics can be found in Appendix D. Understanding the way the modules interact with each other is as important as understanding the actual functions of the modules themselves. All data routing is carried out over the data routing network as previously discussed.

*4.1.2 Test Circuit Creation.* Designing test circuits for implementation on the DRFPGA is significantly different from the design using standard methods such as RTL or VHDL. Since the DRFPGA uses proprietary hardware to implement the functions, all designs must be tailored to the DRFPGA layout. The functionality of the test circuits is implemented in the DRFPGA LUTs. As discussed previously, the LUTs can implement any binary combinational 4-input, 1-output digital operation. Although the LUT itself operates asynchronously, synchronous functions can also be implemented using the flip flops and latches within the CLBs.

Routing signals between functional elements is performed via CLB internal routing hardware. The RTL schematic for a single CLB can be found in Appendix A. Once data exits the CLB, the routing is up to the routing network discussed in Chapter 3. The routing network permits any single CLB to provide input or accept output from any other CLB, although there are limits on how many CLBs may communicate across the routers due to limited physical busses. Due to the fact that the DRFPGA is implemented as a VHDL model, limitations on the physical travel of signals across the data network are non-existent.

*4.1.3 Adder.* A common component in a wide variety of digital systems is the binary adder. The adders used in this research are ripple carry adders, which significantly reduces design and debug time. The adder function implemented in this research takes three inputs ( $A$ ,  $B$  and  $C_{IN}$ ) and provides two outputs ( $SUM$  and  $C_{OUT}$ ). The functionality of the adder is contained within the LUTs (four bits per CLB); for each bit of the adder one LUT output is used to compute the  $SUM$  result

while the other is used for the  $C_{OUT}$  result. The SUM output is provided using the G-LUT whereas the  $C_{OUT}$  is implemented in the F-LUT in the same slice. Each slice implements one full adder, accepting three 1-bit values and providing a 2-bit output. The configuration of single bit adders connected in a multiple bit arrangement is depicted in Figure 4.1. Both 4-bit and 8-bit adders were built for this research. The 4-bit adder consumes one CLB while the 8-bit uses two (being two 4-bit adders chained together). Only the 8-bit adder is explicitly addressed in the testing and reconfiguration phase as the 4-bit adder is included in the 8-bit adder test.

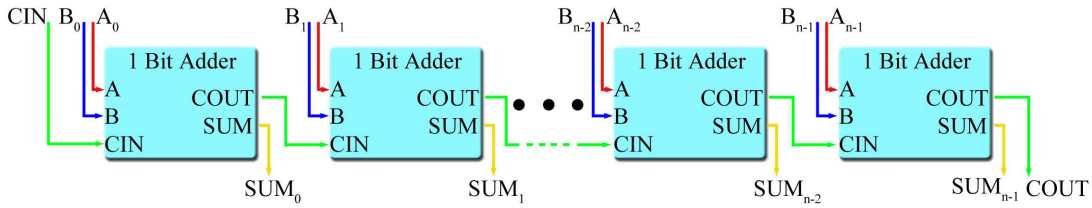


Figure 4.1: Functional configuration diagram for  $n$  bit ripple carry adder. Each 1 bit adder is capable of functioning independently, but can be chained together with other 1 bit adders to create a 2, 3, or  $n$ -bit adder. Each 1 bit adder is implemented in a single slice with one LUT providing the SUM output and the other providing the  $C_{OUT}$  output.

**4.1.4 Counter.** Binary counters represent another fundamental component of digital systems. The counters store a state representing their current count or value. The counters implemented for this research are 4-bit or 8-bit *up counters*. With each clock cycle, the counter implements its value sequentially. Once the stored value has reached its maximum value (all 1's) the counter starts from all 0's and repeats the process. In addition to the clock input, the counters also have an enable signal that can either instruct the counter to increment the value (enable = 1) or hold the current value (enable = 0) and a reset signal that instructs the counter to return to an all 0 state. Similar to the adder, each CLB implements four bits of the counter. In its simplest design, each bit ( $x$ ) in an  $n$ -bit counter must consider the previous values to determine what value it must assume on the next clock cycle. For example, bit three must consider the values for bits zero, one, two, and three to determine what

the next value should be. Given the limited inputs available on the CLBs, this design functions in a manner similar to the ripple carry adder by generating a signal for each bit indicating the values of all the previous bits. The current bit will only toggle if every previous bit is 1; therefore, there is a signal passed through the bits that indicates if every previous bit is 1. The configuration of the counter is graphically represented in Figure 4.2.

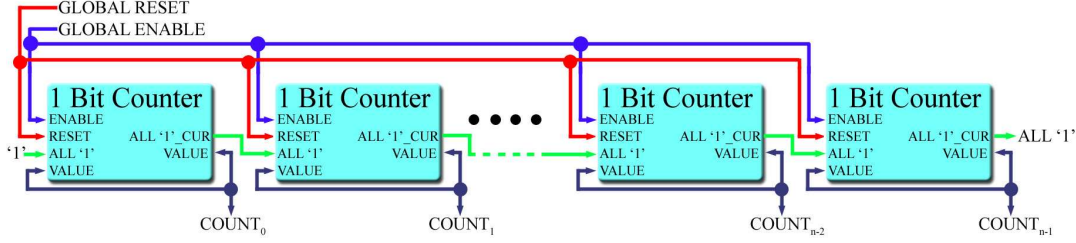


Figure 4.2: Functional configuration diagram for n-bit counter. Each 1 bit counter is capable of functioning independently, but can be chained together with other 1-bit counters to create a 2, 3, or n-bit counter. Each 1-bit counter is implemented in a single slice with one LUT providing the COUNT output for that bit and the other providing the *All 1* output for the current bit. The *All 1* output is high if the current bit and all the previous bits are 1. This indicates that the next bit should toggle at the next clock cycle. The final n-1 *All 1* signal is not used unless the counter is chained to another counter to make a larger unit.

**4.1.5 Comparator.** Comparators provide a means to determine the relationship between two binary numbers, i.e.  $A < B$ ,  $A > B$ , or  $A = B$ . For this research effort, a simple comparator was constructed to determine whether two binary numbers are identical. The circuit is implemented in two different configurations to demonstrate the use of functional replacement. In both configurations, the comparator function is broken down into two stages. In the first stage, input bits are compared to one another and the result of comparing the bits (1 for a match 0 for different values) are passed to the second stage. The second stage compares the results from the first stage and, if any first stage results are 0 indicating a difference in value, then the result of the entire comparison is 0 since at least one of the bits is different. Functionally, the first stage can be viewed as a set of XNORs comparing the input values while the second stage is equivalent to an AND gate ensuring that

all bits for both n-bit input values match ( $A_x = B_x: 0 \leq x < n$ ). For clarification on the gate level implementation, see Figure 4.3.

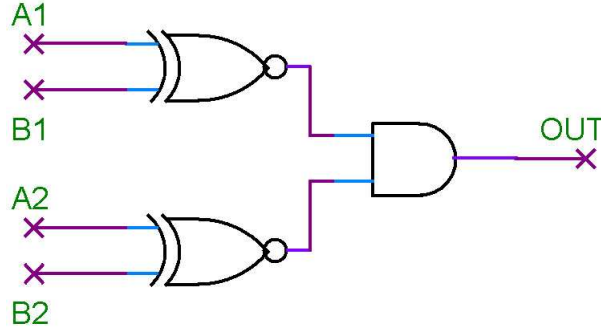


Figure 4.3: Gate level implementation a simple 2-bit binary comparator. The output reflects whether the inputs, A and B, are identical. The output will be 1 for identical inputs or 0 otherwise.

The implementation of the first comparator design within the actual FPGA is constructed such that the stage one functionality (XNORs) are contained in a single CLB while the second stage (ANDs) is contained in a second CLB. Each LUT in the first stage CLB compares two pairs of bits and outputs two values representing each pair's successful or failed match. For example, the first slice takes  $A_0, B_0, A_1, B_1$  as inputs and provides two values,  $A_0 = B_0$  and  $A_1 = B_1$  as outputs through the two LUTs contained in the slice. The outputs are 1 if the values match and 0 otherwise. These output values are fed into the second stage. The second stage is performed in two steps. The first two slices in the second CLB (M0 and M1) each compare the outputs from four of the eight match tests computed in the first CLB. The second two slices in the second CLB (L0 and L1) compare the results from the first Slices M0 and M1 in a hierarchical manner using an AND gate functionality to determine whether all inputs matched. An example of the CLB/LUT schematic is shown in Figure D.3 in Appendix D. The hierarchical relationship between the different stages in a functional diagram are shown in Figure 4.4.

The second comparator design is constructed such that the stage one and stage two functionality are both contained in a single CLB. The first two slices (M0 and

M1) compare four pairs each (for a total of eight pairs needed for the comparator). For example, Slice M0 takes  $A_0, B_0, A_1, B_1, A_2, B_2, A_3, B_3$  as inputs and outputs two values.  $(A_0 = B_0) \cdot (A_1 = B_1)$  and  $(A_2 = B_2) \cdot (A_3 = B_3)$ . The outputs are 1 if the values match and 0 otherwise. These output values are fed into the second stage. The second stage compares the results from the first stage using an AND gate functionality to determine whether all inputs matched. An example of the CLB/LUT schematic is shown in Figure D.4 in Appendix D. The hierarchical relationship between the different stages in a functional diagram are shown in Figure 4.4.

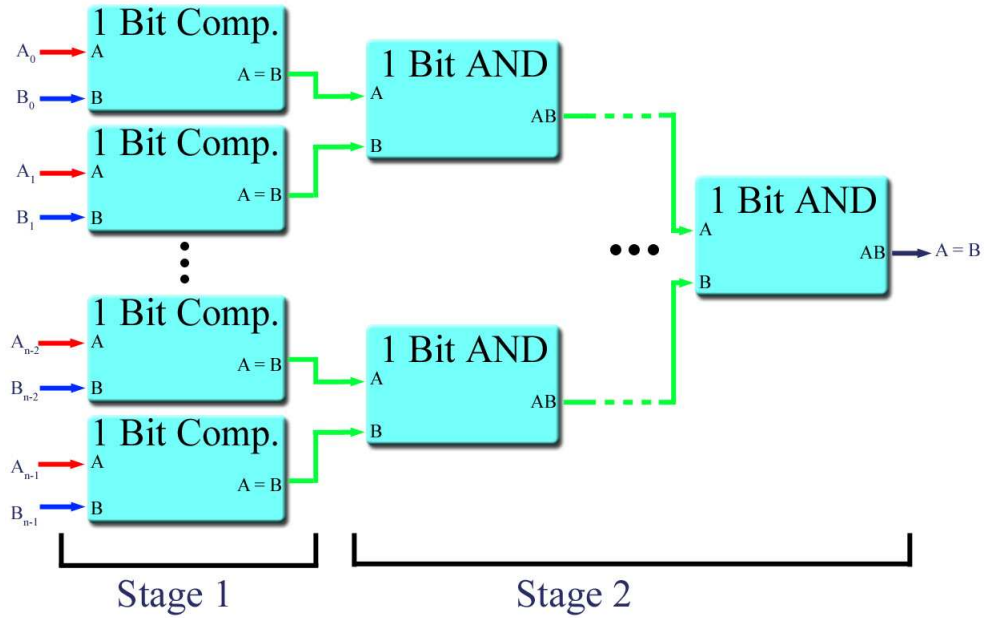


Figure 4.4: Functional diagram demonstrating the hierarchical structure of a binary comparator. The example has single bit comparators in the first stage, matching the equivalent layout of the first comparator configuration. The second, functional replacement configuration contains the equivalent of 2-bit comparators, thus combining the functionality of two LUTs from the original design into a single LUT. Both implementations use 4-bit AND gate equivalents in the second stage, but due to the differences in output lines, the original implementation has three levels of ANDs in Stage 2 as compared to a single level for the functional replacement.

## 4.2 Reconfiguration

*4.2.1 Reconfiguration Overview.* The test circuit bitstreams are implemented on the VHDL model DRFPGA and tested for correct operation. Each test

bench compares the results from the FPGA implementation with the results of a similar function implemented behaviorally. Given the large number of test cases as described in Chapter 3, the tests were ran using the *assert function*. The *assert function* allows the test to be automated and informs the tester of any input combinations that triggered an incorrect or unexpected output.

Once the original circuit is shown to be operating correctly, the reconfiguration process must be tested for every circuit. LUT inversion reconfiguration is performed on the adder and counter test circuits and functional replacement test reconfiguration is conducted on the comparator circuit.

The LUT inversion tests are relatively straight forward. A pseudo-random algorithm, implemented as an XNOR LFSR, is used to randomly select single LUTs to invert. While the current FPGA is 16 CLBs and 128 LUTs in size, the LFSR used provides a 10-bit pseudo-random number. For the adder and counter circuits, each having 16 LUTs, the target is selected by using the lower four bits of the 10-bit value. Once the target address has been generated, the initial inversion is performed on the selected LUT. Following the initial inversion, the selective segment swapping (Phase 2) is executed to complete the overall LUT inversion process. Once the entire process has been completed, the output results are evaluated to ensure the circuit is correctly operating.

The functional replacement tests replace the original functional units with an alternate implementation. For example, the comparator circuit has two implementations that are identical in function, but different in design. The original circuit is reconfigured to implement the second, functional replacement design. This simulates the substitution of a module from the library of modules for replacement. Since the target functional circuit may vary widely in size, the selection process is different from that used in the LUT inversion. The circuit boundaries are determined manually, and a suitable replacement bitstream is programmed into the FPGA to simulate an autonomous design. The replacement module for the comparator is precompiled.

*4.2.2 Adder Reconfiguration.* The LUT inversion test for the adder completed successfully. The initial step is selection of a random (or pseudo random) LUT to initiate the process. As explained in Chapter 3, each LUT relying on the initial LUT’s output will then have selected segments of its contents swapped with neighboring segments. Details on the LUT inversion process are shown in Appendix B. The original circuit was given five test cases to demonstrate the operation. A full test is not required at this point as the circuit was tested for correct operation previously. Once the five test cases had been presented, the circuit was reconfigured. The LFSR provided a pseudo-random binary number of “0101” indicating a selection of LUT-5. LUT-5 is the F-LUT contained in Slice L0 used to compute the  $C_{OUT}$  value for bit-2. For clarification, see Figure D.1. This represents the LUT that will be completely inverted as stated for the Stage 1 actions. Once the initial LUT was inverted, the circuit outputs are incorrect until the second stage has been completed. This attribute provides the basis for the phased defense discussed in Chapter 5. Because the  $C_{OUT}$  value for bit-2 impacts the values for bits 3-7, an incorrect value will adversely affect the adder output on all subsequent bits.

Once the initial inversion has been performed, the LUTs relying on the LUT-5 output must be located and inverted with respect to the rules outlined in the LUT inversion algorithm. LUT-5’s output is connected to the inputs of two different LUTs: LUT-6 (Slice L1 G) and LUT-7 (Slice L1 F). The output is connected to input-3 on both LUTs so the Stage 2 LUT inversion performs a swap of the first eight ( $2^3$ ) bits for the second eight. For example, if the contents of the LUT before inversion were “00110110 11101110”, the resulting contents would be “11101110 00110110”. Following Stage 2 completion, the circuit returns to normal operation.

The entire reconfiguration process took 69 clock cycles from when the inversion LUT network was initialized to when the final LUT was inverted and the correct output was available. For reference, it takes a total of 640 clock cycles to reprogram a CLB using the CLB programming network and a total of over 2500 clock cycles using a serial JTAG interface with a column size of four. Although the LUT inversion

reconfiguration process took 69 clock cycles, the circuit was correctly operating for the initial 23 clock cycles because these are used to address the initial LUT inversion target for Stage 1 and no reconfiguration takes place until the target has been found and the command issued.

*4.2.3 Counter Reconfiguration.* The LUT inversion test for the counter completed successfully. The counter was enabled and allowed to count to 64 as a demonstration of the correct operation. Following the operational phase, the initial LUT for inversion was selected using the four bits from the LFSR, just as in the adder reconfiguration test. Since the same seed is used on the LFSR, this test would target the same LUT if the LFSR was allowed to run the same length of time as the adder LFSR. Therefore, the LFSR in the test was allowed to run for an additional two clock cycles to develop a different number. The target was “0111” or LUT-7 (LUT-F in Slice L1). The schematic of the circuit for identification of the appropriate LUT is shown in Figure D.2. As with the adder, the output of the circuit will be incorrect until the final stage of the LUT inversion is complete.

The configuration for the counter is similar to the adder because the initial target LUT’s output is connected to two other LUT inputs. The major difference is the sequential characteristics of the circuit. The adder is combinational; therefore, the output is solely dependant on the inputs. The counter output is dependant on the inputs and the current values stored in the FFs. For example, with an input of  $en = 1$  and  $reset = 0$  the output can either be 0 (if the current value is 255) or current output + 1. This plays an important part in the determination of how the reconfiguration must be accomplished. If the reconfiguration is performed while the circuit is in operation (i.e.  $en = 1$  and  $reset = 0$ ), the count will be disrupted and the output will be unpredictable. Following the initial inversion, the output values are seemingly random. Once the reconfiguration process is complete, the count will resume correctly, but from whatever value was currently stored during the sporadic operation. To remedy the situation, the count was halted ( $en = 0$ ) for the entire

reconfiguration process and resumed once the process was completed. The issues related to continuous operation during the reconfiguration process are depicted in Figure 4.5, while Figure 4.6 demonstrates proper operation of the circuit during the reconfiguration.

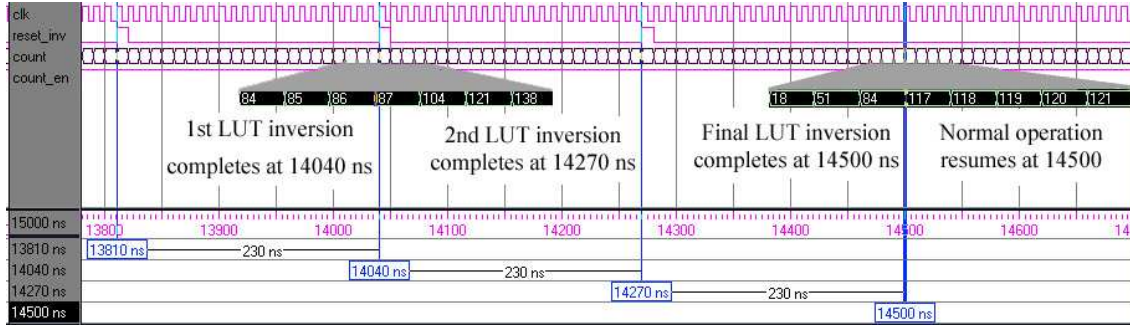


Figure 4.5: Demonstration of the counter output errors due to operation of the circuit during the reconfiguration process. The count value after 87 is a seemingly random value based on the inverted initial target LUT. The output will continue to be incorrect until the reconfiguration process is complete, at which time it will count correctly from whatever value it happens to be when the reconfiguration is complete.

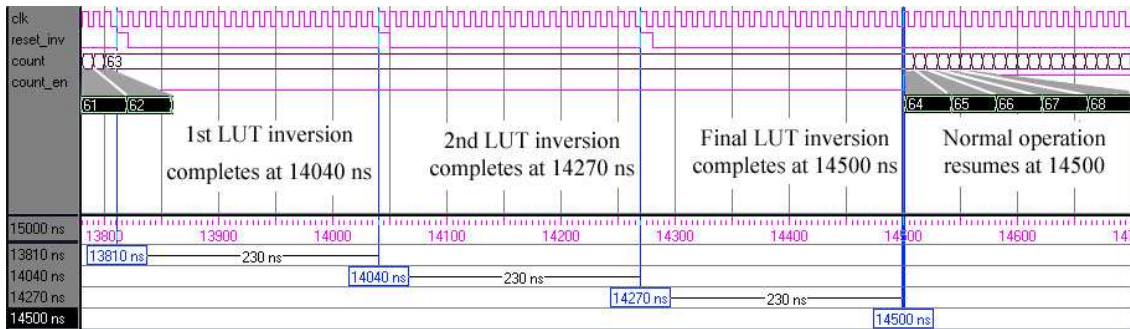


Figure 4.6: Demonstration of the proper method to reconfigure a sequential circuit. The circuit is halted for the duration of the reconfiguration process. Because the LUT inversion does not affect the storage elements (i.e. flip flops), the current value will not change due to the reconfiguration. Once the reconfiguration is complete, the circuit resumes standard operation.

The timing specifications for the counter circuit are identical to the adder given the similar relationship between the target LUTs and the LUTs relying on the target LUT output. The process takes a total of 69 clock cycles, with the first 23 not impacting operation. Although the counter only had to be disabled for 46 clock cycles,

it was disabled for the entire 69 clock cycles to clearly demonstrate the reconfiguration process.

*4.2.4 Comparator Reconfiguration.* The comparator circuit differs from the other two designs because the method used to reconfigure the circuit is the functional replacement method, as opposed to the LUT inversion process used for the adder and counter. The functional replacement reconfiguration completed successfully with the final circuit operating as expected. For the purpose of functional replacement, two schematically different, but functionally equivalent circuits were designed. Each design accepts two 8-bit numbers and provides a 1 as output if the two numbers are identical else the output is 0. The initial design is larger than the second and uses different pins for input and output.

The first step in performing functional replacement reconfiguration is to locate the boundary of the function to be replaced. In this case, the boundary includes CLB-0 and CLB-1. The replacement design will occupy CLB-0 only and will not use CLB-1 pins for output. Due to the shorter chain of LUTs for the input to output travel, the results of the replacement improve the performance and provide output obfuscation for security. The second design occupies less area and has less delay than the original given a shorter data path. For security, the second design relocates the output value consistent with spatial obfuscation. In this manner, any adversary observing the output in a black box attack or general analysis would not receive the correct output values following the reconfiguration. Any authorized system would need to have knowledge of the inner workings and original circuit to anticipate the output obfuscation.

### **4.3 Test Results**

*4.3.1 Test Results Overview.* Every test completed successfully with regards to correct circuit operation. In addition to ensuring that the reconfiguration does not adversely affect the correct operation of the circuit, the affect of reconfiguration on

circuit operation efficiency must be considered. The test circuits operate with a negligible impact due to reconfiguration time overhead. The two circuits reconfigured using the LUT Inversion method realized a 13.48% and .5284% loss in operational efficiency for the adder and counter circuits respectively. Discrepancies between the two circuits are due to the reduced time between reconfigurations for the counter circuit. The functional replacement process resulted in increased performance due to the consolidation of Comparator circuit functions.

The DRFPGA performed favorably when compared to the serial JTAG method of reconfiguration implemented in conventional FPGA. The LUT inversion process imposed the lowest penalties in time and power costs for reconfiguring circuits. This is because the LUT inversion requires transmission of only 23 bits of data as opposed to 626 bits for the functional replacement method. The functional replacement method, on the other hand, displayed lower penalties in power and speed than the JTAG implementation when performing reconfigurations consistent with security DRFPGA operations. It should be noted that the JTAG method is more efficient in power and time for any CLB count that is a multiple of four because it does not incur the addressing overhead of the functional replacement method. This value would change for future, larger implementations of the DRFPGA to further favor the functional replacement method over the JTAG due to a courser JTAG resolution for larger FPGAs.

The addition of the functional replacement and LUT inversion hardware imposes an area increase of less than 6%. This represents a negligible increase considering the added functionality. A DRFPGA identical in size to a conventional FPGA with 60,000 CLBs would have 55,000 CLBs for operational functions.

*4.3.2 Circuit Operation and Performance.* Each circuit was tested in two variations: pre and post reconfiguration. The initial and reconfiguration programming was performed using the CLB addressable network. Each circuit is strictly diagramed to provide an accurate portrayal of which individual FPGA components are used. This

information dictates the delay and speed attributes of the circuit. All attributes were derived using the Cadence® software suite. The longest path through the configured circuit was determined and every component along the path is included in the results. Because the FPGA allows loop-back connections, the software must be specifically directed on which components are under analysis. These components are evaluated individually and the results combined for the final value.

In addition to measuring the performance of the circuit itself, it is also necessary to decide on the time between reconfigurations. The time for the adder is decided by taking the maximum amount of time to discover the functionality and dividing it by 10. This equates to reconfiguring the circuit one tenth of the way through any black box analysis attack. The sequential circuit is significantly more complicated to determine given a black box approach. For each possible combination (256 total) of FF values, testing all possible input values is imperative. For the sequential circuit, the time between reconfigurations is reduced to half of the total time to evaluate. This equates to reconfiguring the circuit halfway through the black box test. Finally, for the comparator circuit, the reconfiguration takes almost 100 times the clock cycles that the LUT inversion needs due to the need for bitstream transmission. Therefore, the time is half of the total time to discover the circuit.

Table 4.1: Specifications for the test circuit pre and post reconfiguration. Post reconfiguration operation includes a recurring reconfiguration. The impact of the recurring reconfiguration is factored into the total operations per second.

Adder Test Results					
Test Run	Size(LUTs)	Speed (MHz)	Time( $\mu$ s) Reconfiguration	Cycles Between Reconfigurations	Operations Second
Original Circuit	8	32.17	NA	NA	32.17M
LUT Inversion	8	32.17	2.14	13,107	32.00M
Counter Test Results					
Test Run	Size(LUTs)	Speed (MHz)	Time( $\mu$ s) Reconfiguration	Cycles Between Reconfigurations	Operations Second
Original Circuit	8	74.92	NA	NA	74.92M
LUT Inversion	8	74.92	0.9210	512	64.82M
Comparator Test Results					
Test Run	Size(LUTs)	Speed (MHz)	Time( $\mu$ s) Reconfiguration	Cycles Between Reconfigurations	Operations Second
Original Circuit	7	44.83	NA	NA	44.83M
Functional Replacement	3	74.92	85.42	32,768	60.28M

The final values indicate the LUT inversion process does not significantly impact the system performance. Performance of the LUT inversion algorithm, however, does impact the system as seen in the adder and counter operations per second values. This is to be expected as alterations to the LUT contents will not add or subtract from the components necessary in the data path, but any process taking clock cycles from the primary operation will affect the overall performance. The primary use of the LUT inversion is to provide a means of obfuscation of the circuit bitstream. Additionally, the LUT inversion may be used in conjunction with LUT merging and splitting algorithms to provide an evolutionary basis for system improvement. The performance and capabilities of the LUT inversion algorithm meet the expectations for this research effort. Because the LUT inversion process actually modifies the LUT contents and bitstream, it also defends against readback or bitstream theft vulnerabilities if the implementation is phased. If the first phase of a LUT inversion is in operation while the second phase is in operation elsewhere in the circuit, then the overall result is a bitstream that will not operate correctly without the exact reconfiguration settings of the original circuit.

The functional replacement process impacts both the security stance and performance of the circuit. In this example, the replacement circuit design is both faster and requires a smaller area than the original design. The replacement circuit occupies less than half (42.8%) of the original design area. It is important to note that even with the recurring reconfiguration, the overall performance in operations per second is increased. In addition to saving space and time, the new design also increases the security of the circuit by disguising the operation through spatial obfuscation of the output. The initial design had the correct output on bit 20, bus row0\_CLB1\_BUS\_1.OUT. This was changed to bit 16 bus row0\_BUS\_0.OUT on the reconfigured circuit. Not only is the valid data output relocated, but the original output location now provides incorrect data. In this respect, the reconfiguration has relocated the output (spatial obfuscation) and created a decoy circuit that will not match the function output.

*4.3.3 DRFPGA Programming Network Performance.* One of the key features to consider regarding the implementation of additional hardware is the cost or gain in power, area, and speed over the original design. The Cadence® software suite was used to simulate the design fabrication given 90 nm technology so a new reconfigurable design can be conservatively compared to the original hardware. Two different power attributes are measured: active and passive. The active power measurement seeks to identify the power usage while the system is being (re)configured while the passive measures the power leakage from the additional hardware during circuit operation, but not being configured or reconfigured. The speed of the device is measured by taking the inverse of the maximum delay for a signal to travel from an input to an output. In addition to the delay measurement, speed is also determined by the total amount of steps necessary to perform a desired function. The area will be measured as the difference or ratio of the original FPGA area and the DRFPGA area.

*4.3.3.1 DRFPGA Power Usage.* For the programming network, the power measurement considers the JTAG capable control registers for the original design and the complete control register programming network for the DRFPGA design. Since the CLBs are not affected by the change in the CLB addressable array, they are not considered for the measurement of the programming network power usage. To properly figure the power ratio of the programming network over the standard programming setup, the total time necessary to program using each method must be considered along with the power usage of the method in use. The standard method requires all control registers within a column to be in operation through the entire serial transmission of the four CLB bitstreams. Given that each CLB requires 640 clock cycles to program, the total time amounts to 2,560 clock cycles to reconfigure a column of CLBs. The programming network only requires one control register to be in operation through a single CLB programming time span (640 clock cycles) in addition to the programming network operating throughout the entire programming period. Assuming that only one column of CLBs (four control registers) is used

to perform a single CLB reconfiguration for the original JTAG method, the new CLB addressable network consumes 8.29% of the power used by the standard control register method while the circuit is reconfiguring a single CLB. There is, however, a 2.12% increase in the leakage power drawn by the device due to the addition of the programming network. Since the design is a reconfigurable array, and will be in a configuration state for most of the operation time, it is the desired outcome.

One case in which the power is equal or greater is any time the programming network is used to load more than six CLBs. The relationship between power usage and the number of CLBs configured are depicted in Figure 4.7. Because of the additional hardware implemented to allow granular reconfiguration, the overall efficiency per CLB is lower than the original design for larger reconfigured CLB counts. Most notable are the multiples of four. Because the overall efficiency per reconfigured CLB is higher for the JTAG implementation, any reconfiguration involving entire columns will not realize benefits using the CLB programming network. Dynamic reconfiguration for security applications typically will not target entire columns of CLBs; therefore, the power usage for targeted applications still favors the CLB programmable network. Additionally, the efficiency of larger CLB arrays will favor the programmable network due to the coarser granularity of the JTAG programming selection. For example, a 16 x 16 CLB array would require approximately 4-W to reconfigure anything less than 17 CLBs. This gives the programming network the advantage on power usage up to a configuration of 12 CLBs. Eventually, the goal is to fabricate the DRFPGA on a scale comparable to *Virtex4*<sup>®</sup> designs of 30,000 CLBs. This size would definitely favor the programmable network for granular security minded reconfigurations.

The LUT inversion network is expected to consume less power than either the JTAG or programming network method of reconfiguration because it broadcasts a 4-bit opcode instead of the 626-bit bitstreams used by the other methods. This comes at a cost of reduced capabilities, i.e. the LUT inversion network is limited to issuing LUT commands and cannot reroute data within CLBs or affect the FFs within the CLBs. The concept of the LUT network is similar to the CLB programming network

since a single element, in this case a LUT, is targeted for reconfiguration. The primary difference between the two is the LUT issues a 4-bit LUT command as opposed to a 626 bit bitstream. This significantly reduces the power usage to reconfigure a CLB, as evidenced by Figure 4.7.

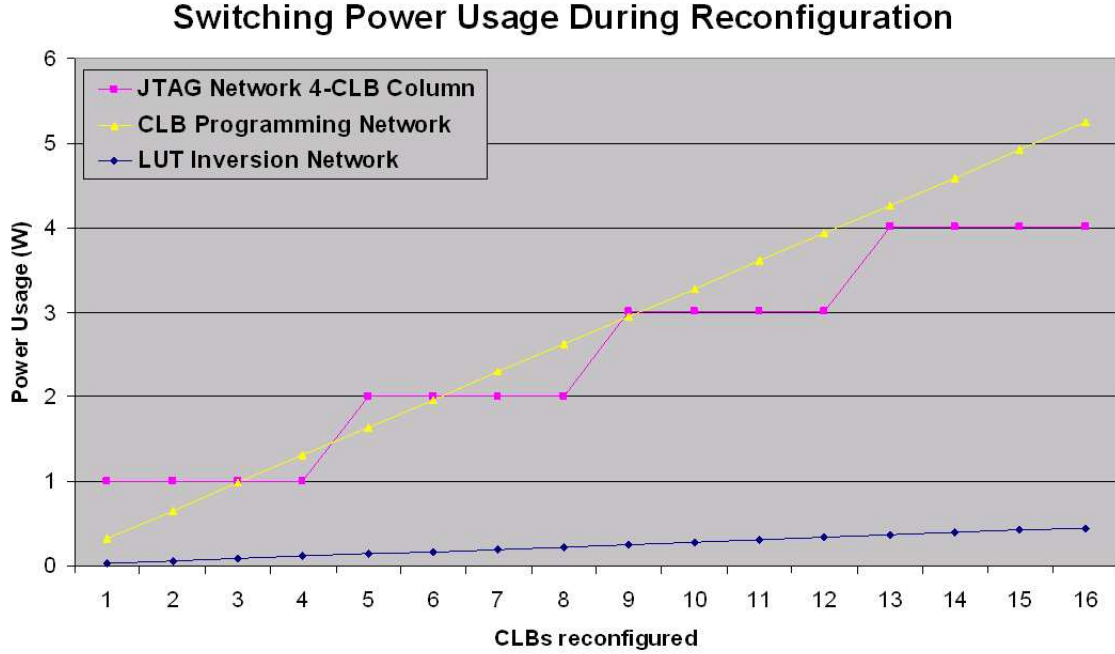


Figure 4.7: The programming network is more efficient for reconfigurations involving less than four CLBs as well as five or nine CLB counts. Given the expected low CLB reconfiguration count for security minded reconfigurations, the programmable network displays favorable characteristics as far as power usage. The LUT inversion network is preferable to the other methods for performing LUT inversions, but is more limited in functionality

It should be noted that to reconfigure an entire CLB, the LUT inversion process must actually reconfigure eight LUTs. Programming the LUTs incurs an overhead of 23 clock cycles comprised of 16 for addressing the target LUT, four for the LUT command, and three for the LUT to execute the command. The total time to address a single LUT for reconfiguration in a column selectable JTAG configuration is 2,560 clock cycles. It is evident the LUT inversion method requires significantly less time in terms of clock cycles, and consumes less power. The maximum power used by the LUT inversion process to perform LUT inversion on every LUT within the FPGA

is less than 0.5-W. This makes it the clear choice when attempting to perform a LUT based reconfiguration method and highlights one of the advantages of granular reconfiguration.

*4.3.3.2 DRFPGA Area Usage.* The overall size of the DRFPGA with the programming network or LUT inversion network will be larger than the original design. The absolute size and area increase attributable to the added reconfiguration hardware is summarized in Table 4.2. Although the LUT inversion network is very similar in design to the programming network, the overall area impact for implementation is higher due to the use of hardware in the CLBs used to route and store the LUT inversion commands.

Table 4.2: Area values for the original circuit, LUT inversion network, and programming network

Area Results		
	Size( $\mu m^2$ )	Area Increase
Original Design	468,462	NA
LUT Inversion Network	16,955	3.62%
Programming Network	9,845	2.10%
LUT Inversion Network and Programming Network	26,800	5.72%

*4.3.3.3 DRFPGA Time Usage.* In addition to the power and space requirements, it is important to consider the overall impact to the reconfiguration time resulting from the implementation of the programming network. As discussed in the power usage section, the programming network requires less clock cycles for any operation where fewer than four CLBs are reconfigured. The synthesis results obtained through the Cadence® software indicates the CLB programming network can operate at a higher clock cycle than the control register hardware. To reconfigure a CLB with the CLB programming network, the control register is used to store

the values in the CLB; the results are compiled using the most conservative operating frequency for both the programming network and the JTAG reconfiguration methods. The operating frequency for the reconfiguration hardware is conservatively set at 350 MHz. The time required to perform reconfigurations for the JTAG and CLB programming network are shown in Figure 4.8. The values displayed are in clock cycles to provide a clear comparison. To put the overall time in perspective, at 350 MHz a complete configuration would occur in less than 34  $\mu s$ .

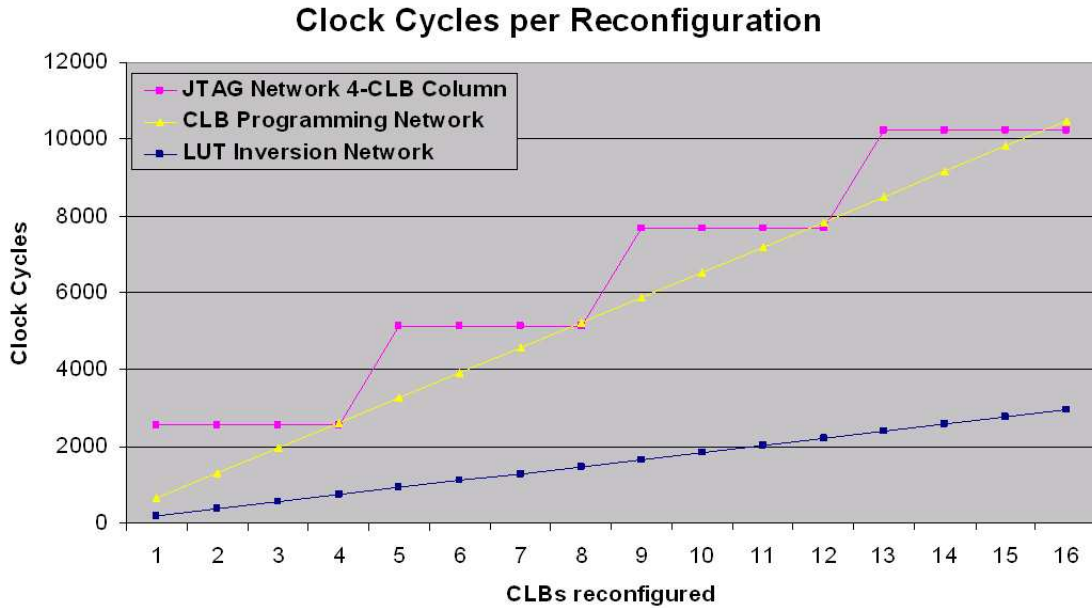


Figure 4.8: The programming network is more efficient for reconfigurations on any CLB count that is not a multiple of four. Given the expected low CLB reconfiguration count for security minded reconfigurations and the low percentage of CLB counts that are evenly divisible by four, the programmable network displays favorable characteristics as far as time to reconfigure. The LUT inversion network is preferable to either of the other methods for performing LUT inversions, but is more limited in functionality

The programming network displays clear advantages for CLB counts that are not multiples of four. The JTAG method reconfigures columns of four and it maintains advantages for any multiple of four due to the addressing overhead imposed by the programming network. The overhead amounts to 14 clock cycles per CLB and is

not enough to offset the disadvantages of the JTAG method except for the few cases discussed. For a larger design, the JTAG advantage would occur even less frequently. In general, the JTAG method incurs less time overhead any time the total number of CLBs to be reconfigured is an even multiple of the column size for the FPGA. For the 16 x 16 FPGA example mentioned previously, the JTAG would reap advantages in time for every value of  $x$  such that  $x = 16^n$  with  $n$  being any integer.

The overhead for the LUT inversion network is even lower given the substantially smaller (23 vs 626) amount of data bits necessary to perform the reconfiguration. The overall time performance numbers for the LUT inversion network in comparison to both the JTAG and addressing networks is depicted in Figure 4.8. As with the addressing networks, the LUT inversion network is capable of higher operating frequencies than the JTAG network but is purposely limited to the conservative estimate of 350 MHz for ease of comparison. There are no cases where it would be more efficient to perform a LUT inversion process using either other method. It should be noted the values in Figure 4.8 represent the clock cycles necessary to reconfigure an entire CLB (eight LUTs).

#### **4.4 Summary**

Results from testing the DRFPGA verify that the added hardware for functional replacement and LUT inversion offers a power and time benefit over the conventional JTAG programming for security oriented operation. The most efficient means of performing reconfiguration is through the use of the LUT inversion network. The other methods of reconfiguring, JTAG or functional replacement, offer more functionality than the LUT inversion network. Adding the hardware supporting dynamic reconfiguration also adds area to the overall DRFPGA design. The area penalty imposed by the added reconfiguration hardware is negligible at less than 6%.

## V. Analysis and Conclusions

This chapter discusses the analysis of the results from the tests implemented in Chapter 4 and provides conclusions based on the analysis. The tests have verified that the proposed DRFPGA platform yields a reliable method of implementing proprietary designs. The CLB programming network is significantly more efficient in time and power usage for reconfiguring the circuit for most situations related to security minded dynamic reconfiguration. The addition of the CLB addressable network does not render the original serial loading via JTAG boundary scan unusable in any fashion; therefore, either method can be used to program the DRFPGA, allowing the user to reap the benefits of both implementations.

In addition to providing a basis for reliable and efficient dynamic programming, the DRFPGA design also enables users to secure designs against bitstream theft. The tests validate the implementation of a phased defense using the LUT inversion algorithm to obfuscate the bitstream. The concept behind the algorithm is key in understanding the phased process, which can be substituted to yield similar results. The functional replacement method of reconfiguration provides a means to protect the design through input/output obfuscation.

### 5.1 *Bitstream Protection*

As presented in Chapter 2, the bitstream is a vulnerable and valuable target for adversaries seeking to acquire FPGA implemented designs through illegitimate means. The defenses in place for existing FPGA platforms can be circumvented and do not adequately protect proprietary or critical circuits. By implementing the phased defense system of LUT inversion, the designs implemented on the DRFPGA render the bitstream inoperable if taken from the device. By overlapping the current Phase 2 with the next algorithm iteration Phase 1, the bitstream itself, through the configurations stored in the CLBs, is never accurate. As long as the output for the CLBs being reconfigured are not being used by other CLBs during the reconfiguration

process, the phased defense will not have an adverse effect on the accuracy of the system output.

Figure 5.1 demonstrates how the phased defense is used to maintain an inaccurate bitstream. A relatively small circuit is used to demonstrate the process on the test circuits visually is not reasonable and will not convey the information. The sample circuit is shown in six discrete stages of reconfiguration. The original circuit stage shows the initial circuit program. Each box represents a two input single output LUT with the stored values under the LUT label. The top input represents input-0 and the lower input-1. Each reconfiguration is performed in two phases, as outlined in the previous chapter and in Appendix B. During the second phase of an inversion, the next inversion begins its first phase. Since the output of any LUT is incorrect as of completion of Phase 1 (before completion of Phase 2) and there is always at least one LUT completing Phase 1, the bitstream is always incorrect. For the example, the reconfiguration is halted after four iterations (after iteration D) to demonstrate that the final circuit is correct. One drawback to working in the LUT realm is that the circuit cannot be translated directly back to a gate level implementation. Before the reconfiguration, each LUT within the example circuit can be expressed as a single gate. After the reconfiguration, LUT-6 and LUT-8 can no longer be expressed as single gate functions.

In order to properly interpret the results of the circuit, a user must have knowledge of the initial circuit content and the order, timing, and current state of the circuit. If any of this information is not available, then the current circuit contents cannot be positively determined. Since the LUT inversion algorithm's strict adherence to a set of rules, a user with order and timing information can determine which outputs are valid and which outputs are not. The information may be presented as an initial circuit schematic (interconnected LUTs), LUT inversion phases start every 10ns, and LUT reconfiguration order based on an 8-bit LFSR with key of 01001100. While this particular information does not pertain to the circuit below, it is an example of the information necessary to properly determine the accurate output of the circuit.

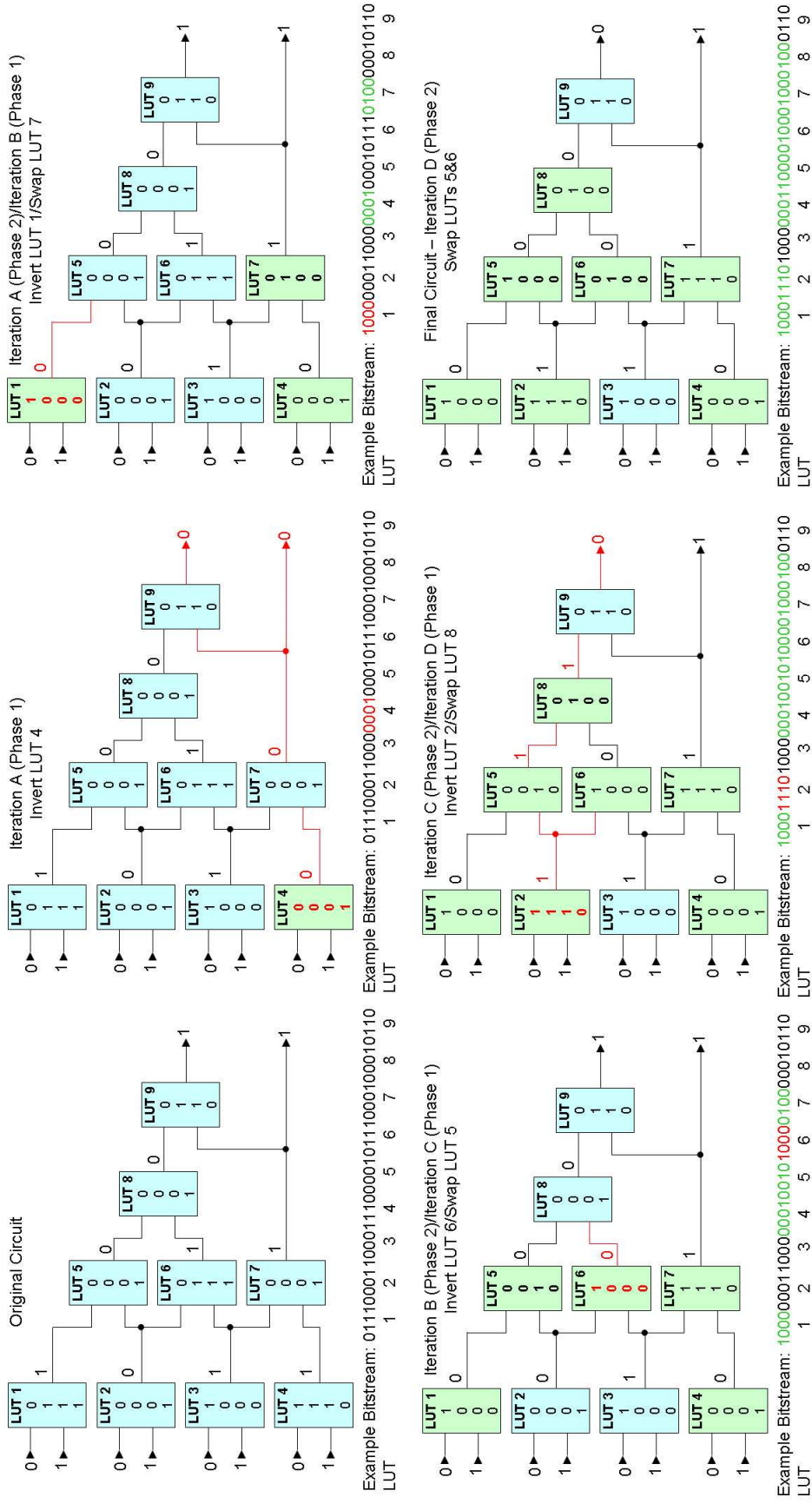


Figure 5.1: Demonstration of the phased defense method of obfuscating the bitstream. After each step, the bitstream is rendered unusable as is evidenced by the red bits. The only way to correctly interpret the output is to have previous knowledge of the initial circuit state and the progression/timing of inversions performed.

## 5.2 Reverse Engineering Defense

In addition to the threat of system compromise via bitstream theft, there also exists the risk of adversarial analysis and tampering using reverse engineering techniques such as black box or power analysis. The functional replacement reconfiguration has shown proficiency in obfuscating the output of a circuit. While the functional replacement method can be used to place modules that are identical in input and output locations to maintain functionality, the output and input can be relocated on the replacement module to provide the obfuscation.

The higher-level abstraction for the functional replacement as compared to the LUT inversion (CLBs instead of LUTs) provides the means to alter the input and output routing. Figure 5.2 shows an example of the actual test comparator pre and post reconfiguration.

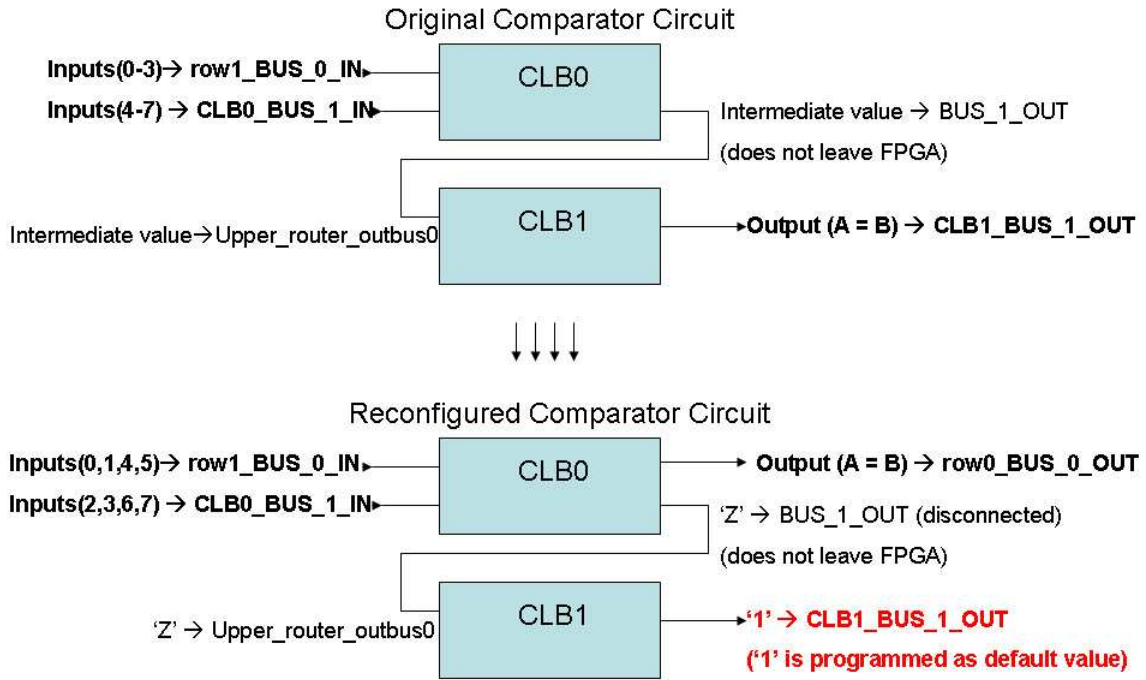


Figure 5.2: Demonstration of the functional replacement method of obfuscating the input and output of a given circuit. The original circuit is replaced with a smaller module with differing input and output locations.

The original circuit uses two CLBs to implement a comparator function. The inputs are divided between two different input buses, with the lower four bits from both A and B placed on a bus together and the upper four bits placed together on a different input bus. The output is provided as a single bit indicating whether A equals B (1 if they are equal, else 0). Since the functionality is divided between two CLBs, there is an intermediate value passed from the first CLB to the second. Following the functional replacement, the input bits are moved between the two input buses. Where the bits were divided evenly by input value rank, they are divided in a seemingly random configuration (0,1,4,5 input bit positions on one bus and 2,3,6,7 input bit positions on the other). In addition to moving the input value locations, the output has been relocated from an output bus on the second CLB to a different bus on the first CLB. The bus that originally carried the intermediate value is now disconnected and has the value Z for high impedance on all the bus bits. The second CLB is programmed to output a 1 if it receives a Z as input. The second CLB is now disconnected and any adversary monitoring the circuit will receive a valid, if incorrect, value.

Interpreting the results of a functional replacement reconfigured circuit is more difficult than the LUT inversion reconfiguration. The functional replacement process is not a clearly defined algorithm, but more of a method of using the CLB programming network to target specific CLBs for reconfiguration. Therefore, a user would need to have knowledge of the replacement modules, location of replacement, and timing of replacement. The timing and location can be compared to the order (identifying which LUT or module is being reconfigured) and timing as discussed in the LUT inversion discussion above. The replacement module information, however, would have to include every module that could possibly be used to replace an existing function. This adds complexity to the process, but also provides a substantial amount of flexibility for creating various obfuscated functions.

### 5.3 *Circuit Protection Effectiveness*

Although the algorithms and concepts provided have a wide variety of applications throughout the reconfigurable design realm, the focus of this research is the applicability towards protecting circuits implemented on the DRFPGA. For each vulnerability discussed in Chapter 2, there is a method to address it using LUT inversion, function replacement, or both.

*5.3.1 Bitstream Interception.* Even with the encryption employed for the transmission of bitstream data from off-chip storage to the FPGA, there is still the chance that an adversary could compromise the bitstream if they were able to capture the information during transmission. If the adversary were able to obtain the encryption key from the FPGA device, then compromise of the bitstream information would be nearly certain.

To combat the theft and use of a bitstream from the proposed DRFPGA, a programmer can implement the LUT inversion algorithm to provide a mangled bitstream. If the stored bitstream is in a state halfway through the reconfiguration of a CLB (completed Phase 1), the bitstream would not be usable on a standard FPGA or a DRFPGA without the knowledge of the order and current state for the reconfiguration. The bitstream itself is keyed to the DRFPGA design and the current reconfiguration state information. Also, the key is dynamic because the current state changes with every reconfiguration. By encrypting the bitstream and including the LUT inversion protection, the bitstream is adequately protected from unauthorized acquisition.

*5.3.2 Fault Injection.* By injecting faulty signals at the circuit inputs, adversaries are able to impact the successful accomplishment of circuit functions. This is particularly evident in functions such as encryption algorithms that require multiple iterations to achieve their full effect. It is not a trivial matter to target functions using fault injection as the adversary must have knowledge on the timing, inputs,

and overall functions that are being targeted. By using the functional replacement algorithm, the DRFPGA device can minimize the potential impact of fault injection techniques. Users can choose to modify the location of input and output pins, denying any adversary a static target for injection faults. Not only can the functional replacement method provide a means of relocating inputs and outputs, but a user could also implement a sequential circuit to replace a combinational circuit, temporally obfuscating the output by shifting it one or multiple clock cycles from where it originally became available.

The LUT inversion algorithm also assists in defending against Fault Injection. Since LUTs are chosen at random for reconfiguration, adversaries have no knowledge of which LUT may be in the process of reconfiguring. Also, if the fault injection is being performed mid-circuit (as opposed to from the input) using probes, then the attacks are more difficult. After Phase 1 of the LUT inversion reconfiguration, a LUT output that was 1 for a given input will now have an output of 0. If the adversary is attempting to insert an incorrect value to influence the outcome of a circuit operation, the inverted LUT will counter that threat.

*5.3.3 Passive Circuit Analysis.* Using passive circuit analysis, adversaries can attempt to map the functions contained within an FPGA design. Passive analysis may include electromagnetic or thermal observation and input or output monitoring. When multiple observations are combined, the composite picture may provide a reasonable understanding of the circuit functionality enabling more malicious attacks. One of the primary requirements for successful passive analysis is a pattern of activity or readings. Patterns are needed to establish a common response to values or timing of inputs. Both the LUT inversion and functional replacement are beneficial in combating passive analysis of circuits by denying the adversaries a static target.

Any process that alters the composition of the circuit is beneficial to countering the threat of successful circuit mapping with passive analysis. The functional replacement process provides an extremely flexible manner in which to modify the makeup

of the circuit. As in the comparator circuit reconfiguration, inputs and outputs can be re-mapped to different locations. Passive analysis can be used to spatially or temporally locate functionality. For either case, the relocation of the circuit input and output pins will render any further analysis useless. For example, if an adversary is monitoring the output of the comparator test circuit to map the functionality, then once the circuit has been reconfigured, they will be monitoring the wrong output. Even if they were to locate the current input pin, they would need to re-accomplish all previous tests as the input pins have been relocated as well. The comparator is a relatively simple circuit having only 16 inputs (eight for A and eight for B) and a single output. For a simple example like the comparator, the task of completely mapping the circuit requires the analysis of  $2^{16}$  or 65,536 different input combinations. While this task is within reach of most adversaries the addition of a reconfiguration that occurs every 16,000 or even 32,000 clock cycles makes the process significantly more difficult. Additionally, electromagnetic or thermal observations also provide misleading data when the core functionality (or the parts that give off the highest readings in both electromagnetic or thermal energy) is relocated to a different physical location within the FPGA. By moving the functionality to different locations throughout the circuit operation, the adversary is left guessing the actual circuit arrangement after each reconfiguration.

The LUT inversion algorithm is more limited in the protection against passive analysis than the functional replacement, but is not without its own advantages. Since the reconfiguration does not seek to alter the location of input and output pins, reconfiguration will not provide the same displacement and spatial obfuscation advantages as the functional replacement system. The LUT inversion does provide the advantages of modifying intermediate signals between inversion LUT phases. If an adversary is attempting to monitor a signal between LUTs, and the first LUT is inverted (Phase 1 of the LUT inversion process), they will unknowingly capture data inconsistent with the correct operations.

*5.3.4 Altered Bitstream Attack.* In the altered bitstream attack, adversaries attempt to modify the operation of an FPGA by mangling the bitstream itself. The target opportunity occurs during transmission of the bitstream from the off-chip storage to the actual FPGA. Modification of the bitstream potentially exposes design, state, or critical encryption data from the device registers.

Although the bitstream must remain somewhat similar to the original design, the functional replacement process can provide defense against the altered bitstream attack. The algorithm must identify the boundary and function of the targeted module(s) for replacement. If the bitstream has been altered beyond a certain point, existing modules cannot be positively identified due to input and output combinations outside of the limits of recognition or a system boundary that has been increased to where it no longer falls within the range accepted during the boundary identification step. For example, if the original circuit contains a 4-bit ripple carry adder and the functional replacement system can identify this adder and replace it with an obfuscated output ripple carry adder or carry look ahead designs, the altered bitstream significantly changes the adder functionality and the altered outputs will defy identification by the system. As well, if the adder is maliciously altered to consume three CLBs instead of two, the replacement process will not consider the entire adder when identifying the boundary without a robust boundary identification system. This situation addresses the importance of implementing an intelligent boundary identification and input/output analysis system as opposed to a system that operates strictly from precompiled modules. If the system is able to identify the altered adder, it can replace the malicious module with a known good implementation denying the adversary information or operation sought through tampering with the bitstream.

*5.3.5 Bitstream Readback.* To exploit the bitstream readback vulnerability, an adversary must first bypass the bitstream security measures implemented on most current FPGAs. An example of a security measure is the security bit on the **Xilinx® Virtex4®**. In order to download the bitstream from the FPGA, the security bit

must be set to allow readback. Chapter 2 discussed methods found to be successful in altering the value of storage bits using radiation and/or optical energy. Therefore, the security bit should not be considered a failsafe method of protecting compromise of the bitstream data.

The LUT inversion algorithm provides a reasonable defense against the bitstream readback attack. Given the phased operation of the algorithm, the bitstream stored on the device always contains incorrect information causing the overall circuit to operate in an unpredictable manner. The random selection of target LUTs combined with the use of a key to initiate the LFSR used to provide the random target deny the adversary a reliable method of gauging which LUT is currently in failure state. Therefore, the downloaded bitstream would not function correctly if downloaded to another FPGA. The key to successful operation is the storage of current state information. Without the state information, an adversary cannot use the bitstream in the event that it is recorded from the device.

#### **5.4 *Summary***

The data collected on the DRFPGA operation confirms the platform is a valid architecture for protecting the operation and security of circuits. Implementation of the methods provided in this thesis alleviate substantial risks posed to FPGA designs.

## VI. Future Research

This chapter provides ideas as to the direction of future research into dynamic reconfiguration for security minded applications. Although the current research has provided a solid, reliable platform supporting dynamic reconfiguration and cellular autonomy, future research will yield a completely self-contained DRFPGA capable of controlling the dynamic reconfiguration with no user intervention.

### 6.1 *Autonomous Secure DRFPGA Platform*

Future research must continually investigate security applications. The DRFPGA requires significant user interaction to perform the reconfigurations. In the future, reconfiguration commands issued through the VHDL test bench files must be handled by on-board computational resources. The Xilinx® Virtex4® provides PowerPC® microprocessor cores on-chip and can run programs created to automate the reconfiguration processes. Once initially programmed, the DRFPGA will require no interaction from a human interface except for updating, reprogramming, or repair.

By containing all resources needed for reconfiguration on-chip, the security of the device is greatly improved over issuing commands from off-chip. Additionally, self contained units can be integrated with tampering detection methods to prevent unauthorized disclosure. Upon tampering detection, critical components can be erased.

### 6.2 *Biological Computing*

The LUT inversion algorithm is one example of the many uses for DRFPGAs capable of cellular autonomy. Evolutionary computing focuses on replicating biological developments in hardware and software. By controlling the function and programming of the DRFPGA at the lowest reconfigurable level (LUTs), the DRFPGA achieves the required granularity. Because the LUT inversion functionality is built-in at the LUT level, it opens the path for any number of functions to be added or substituted relatively easily.

In addition to the current hardware, the addition of cross talk between LUTs (aside from the data network) could also greatly benefit the DRFPGA as an evolutionary platform. Eventually, the completed device could be programmed and left unattended. Given an ability to monitor performance, the potential exists to develop designs that achieve unique, valuable attributes through development outside the strict processes used by human programmers.

Insect swarming simulations are another example of biological computing. Using the granular programming capability of the DRFPGA and relatively simple decision making programs within the LUTs, the DRFPGA can be used to model select aspects of swarm psychology.

### ***6.3 Reliability and Self-healing***

System hardware is not always in an easily accessible location. Devices may be in place deep within the ocean, in remote locations far from civilization, or even in space. In these situations, maintainers are not able to access the systems for repair if a circuit should become inoperable. Therefore, it is important to build reliable designs capable of operating in extreme environment conditions. Current methods of increasing reliability include electromagnetic shielding, remote access for repair, and Triple Modular Redundancy (TMR). Shielding serves to protect the device through physical means designed to limit exposure to harmful radiation. Remote access is often provided via satellite data links, Internet connections, or even dedicated data lines. TMR is a method of increasing reliability using redundant circuits. The circuits process the same input and provide output to a voting logic module. Based on whether or not the outputs agree, the voting logic selects the output that is most common among the three outputs received.

Using the proposed design, the opportunity exists to create a self-healing platform for use in inaccessible locations. The functional replacement method of reconfiguration can be used to replace faulty modules with known-good versions stored on or off chip. In addition, certain modules may be more resistant to external variables

than other, identical functioning modules. In this way, a circuit can be dynamically configured and tailored to a specific environment.

#### ***6.4 Defragmentation and Self-Optimizing***

Hardware implementations may incur performance penalties with regards to fragmented construction and un-optimized designs. Once programs have been downloaded to an FPGA, the design is not automatically measured for performance or efficiency. FPGAs do not have capabilities in place to reconfigure the circuit during operation even if the performance can be measured to identify shortcomings. The process of evaluating operational circuits and reprogramming based on defragmented or optimized programs is time-consuming and, in some cases, not feasible.

The DRFPGA design proposed in this research can be used to perform defragmentation and optimization. With on-board circuit evaluation capabilities, the DRFPGA can be configured as an autonomous, self-optimizing platform. This offers a unique capability not currently available in the field.

#### ***6.5 Summary***

Although this thesis focused on the DRFPGA for security applications, it also provides a framework supporting a variety of other applications. The flexibility of the design comes in its dynamic reconfiguration capabilities as well as the granular programming capabilities. Any design requiring dynamic reconfiguration can benefit from the proposed design methodologies.

## Appendix A. CLB Schematic

The following schematic demonstrates the layout of a single CLB without including the actual data/programming routing information. The CLB represents the fundamental component of the FPGA design.

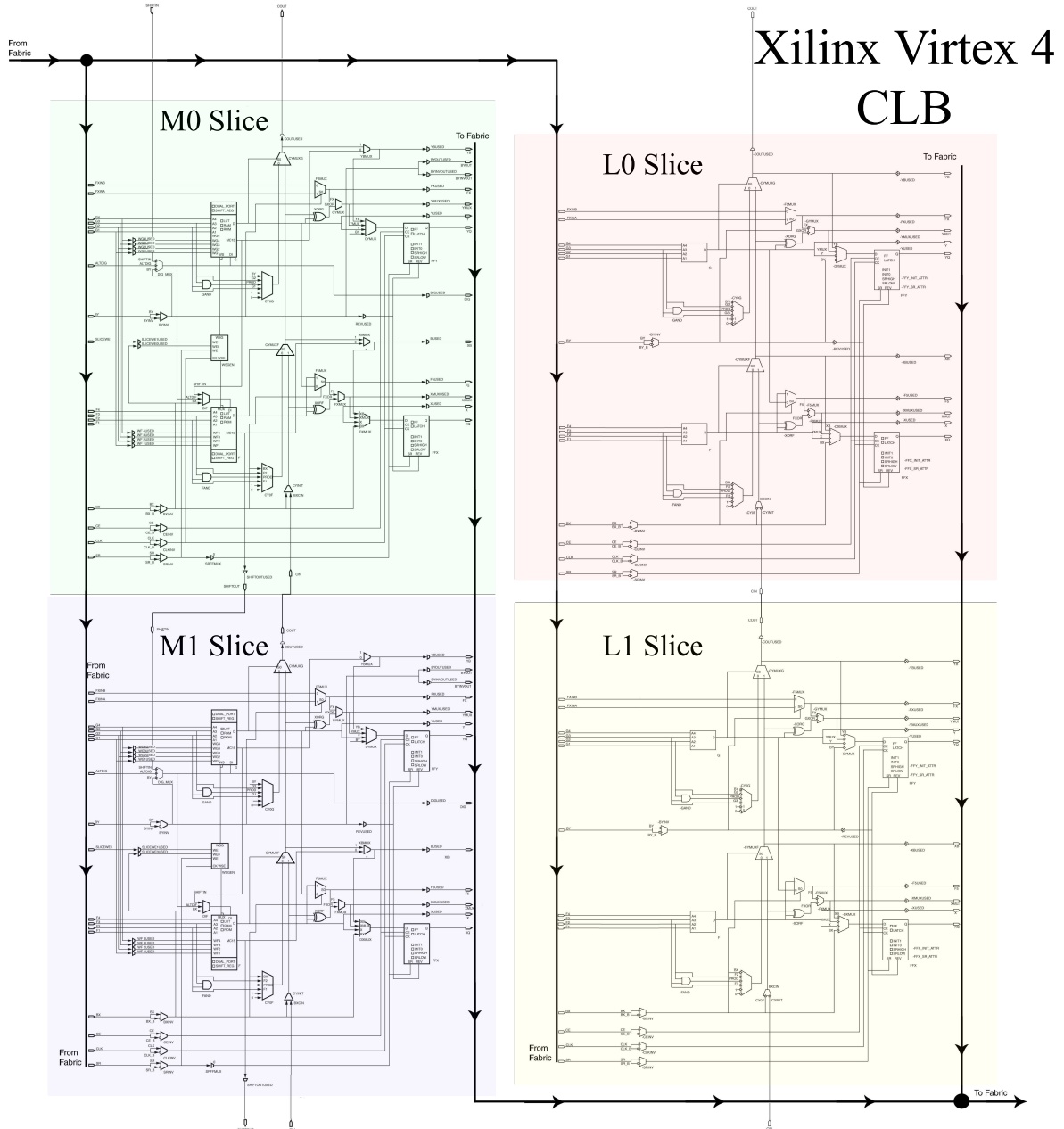


Figure A.1: Schematic of a CLB as based on the schematics of the Xilinx® Virtex4® FPGA [2].

## *Appendix B. LUT Inversion Algorithm*

This appendix documents the steps to perform the LUT inversion algorithm for reconfiguration. Before proceeding with the bubble pushing description, there are some basic attributes and definitions for the circuit that must be outlined.

- All of the LUTs are of the same size with  $2^n$  one bit entries,  $n$  input lines, and one output.
- LUTs are connected in a hypergraph configuration with nodes (LUTs) and vertices (data lines between LUTs).
- LUTs have an absolute identification obtained by levelizing the circuit. A LUT's level is one greater than the highest level LUT feeding it. LUTs connected to the circuit inputs only have a level of 0.
- LUTs are also identified in a relative manner. Given any two LUTs connected together via data lines, the LUT providing the input is LUT- and the LUT accepting the output of LUT- as an input is LUT+.

The bubble pushing step of the reconfiguration is accomplished in a total of four steps split into two stages. The first stage involves selecting and inverting an arbitrarily chose LUT other than a LUT whose output is directly connected to an external devices not include in the reconfiguration.

Stage 1:

1. Step 1: Select  $2^n$  entry LUT within the circuit for initial bubble push
2. Step 2: Invert all entries, 0 through  $2^n-1$ , within the LUT

For the second stage, the following steps must be accomplished for every LUT+ connected to the initial LUT inverted in stage 1. For reference, the initial LUT will be referred to as LUT- and the input line connected to the output of LUT- will be input  $x$  as counted from least to most significant of the input lines (which may be a different line for each LUT+).

Stage 2:

1. Step 3: Given the input,  $x$ , that the LUT- connects to on LUT+, partition the entries in LUT+ into equal segments of size  $2^x$  (for example, if the LUT- output is connected to input 2 on LUT+ then you would partition the entries in LUT+ into segments of size  $2^2$  or 4)
2. Step 4: Exchange the contents of neighboring segment pairs starting with 0,1. For example, segment 0 and 1 exchange contents, then 2 and 3, 4 and 5, 6 and 7, etc.

The following figures demonstrate the LUT inversion algorithm through the reconfiguration of a 7 LUT specialized counter circuit. There are a total of 16 inputs to and 3 outputs from the circuit. The inputs are sectioned into four sets of four inputs each. The goal is to count how many of the four sets have at least two 1's on the inputs. The circuit can be broken down into two different levels, labeled as Layer 1 and Layer 2 in the diagrams. All of the Layer 1 LUTs are LUT-'s as defined in the previous description of the algorithm. Conversely, the LUTs in Layer 2 are all LUT+'s.

As discussed in the process description previously, the next step will involve exchanging neighboring segments of LUT values. The size of the segments is dictated by the input that is connected to LUT-. Since the LUTs in use have four inputs, the largest segment size is  $2^0$  or 8.

The circuit from the example was implemented in VHDL for the purpose of functionally testing the original and reconfigured configuration. The VHDL code can be found in Appendix D. For this test, each of the layer 1 LUTs are inverted followed by the execution of the stage 2 steps on the layer 2 LUTs. The results support the fact that the operation has not been altered due to the reconfiguration. The test results are available in Figure B.5

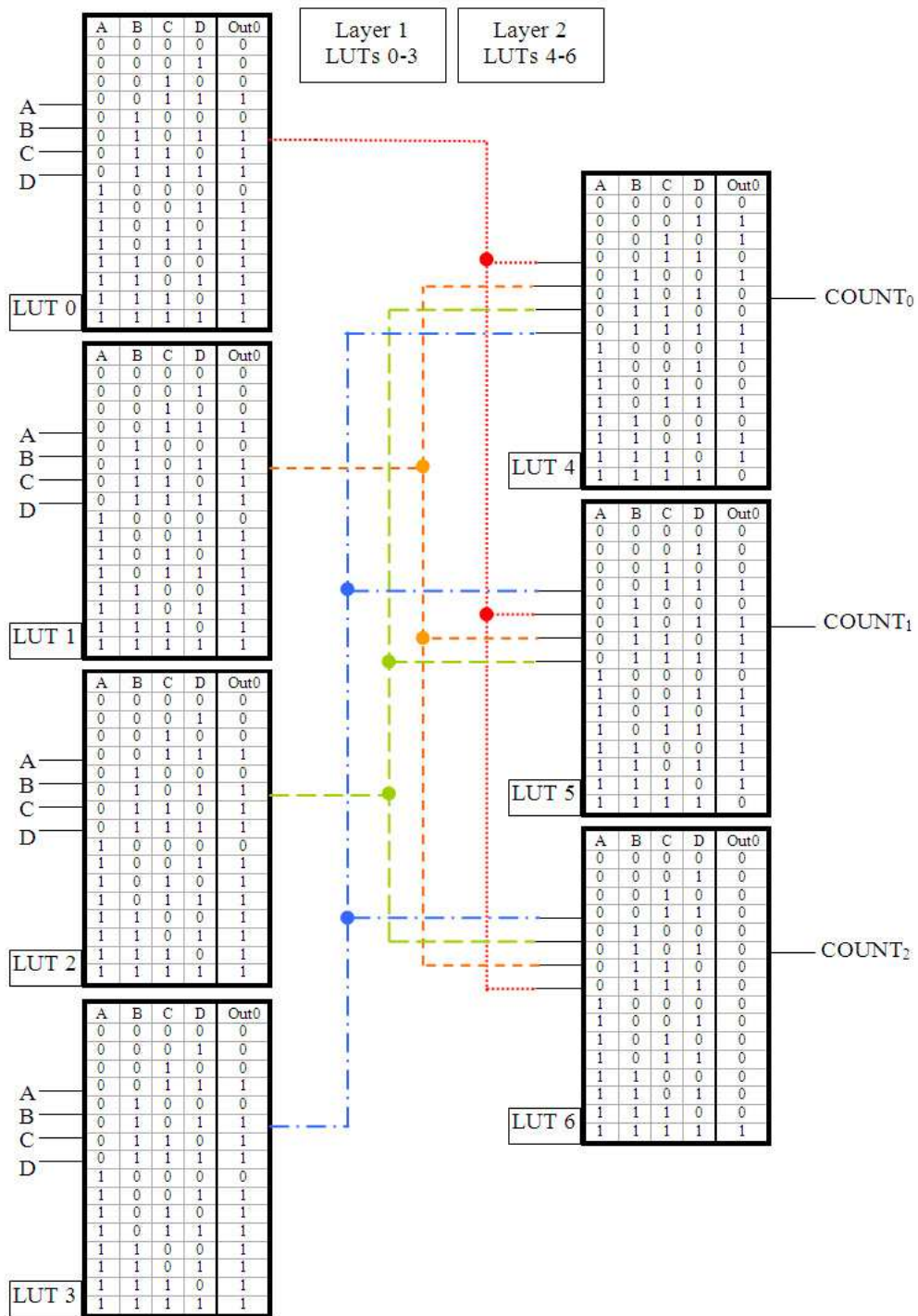


Figure B.1: Sample circuit for LUT inversion demonstration. Layer 1 LUTs are LUT-'s as referenced above whereas the Layer 2 LUTs are LUT+'s. The three bit Count value will express the number of input sets (as connected to the input LUTs in layer 1) that have at least two "1" values on the input.

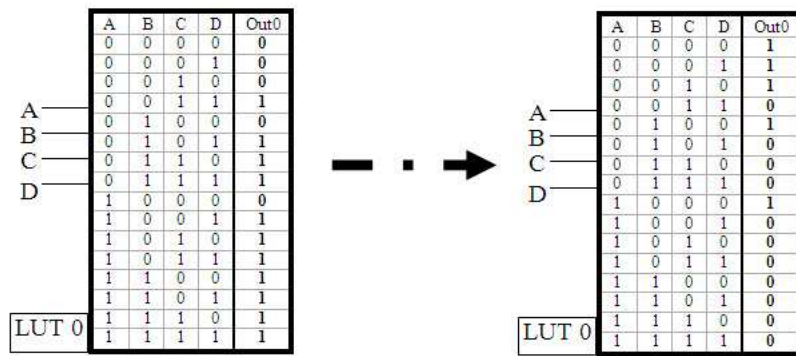


Figure B.2: Demonstration of steps one and two of the LUT inversion algorithm. LUT 0 is chosen as the target LUT for inversion and the values are inverted (step 1 and 2).

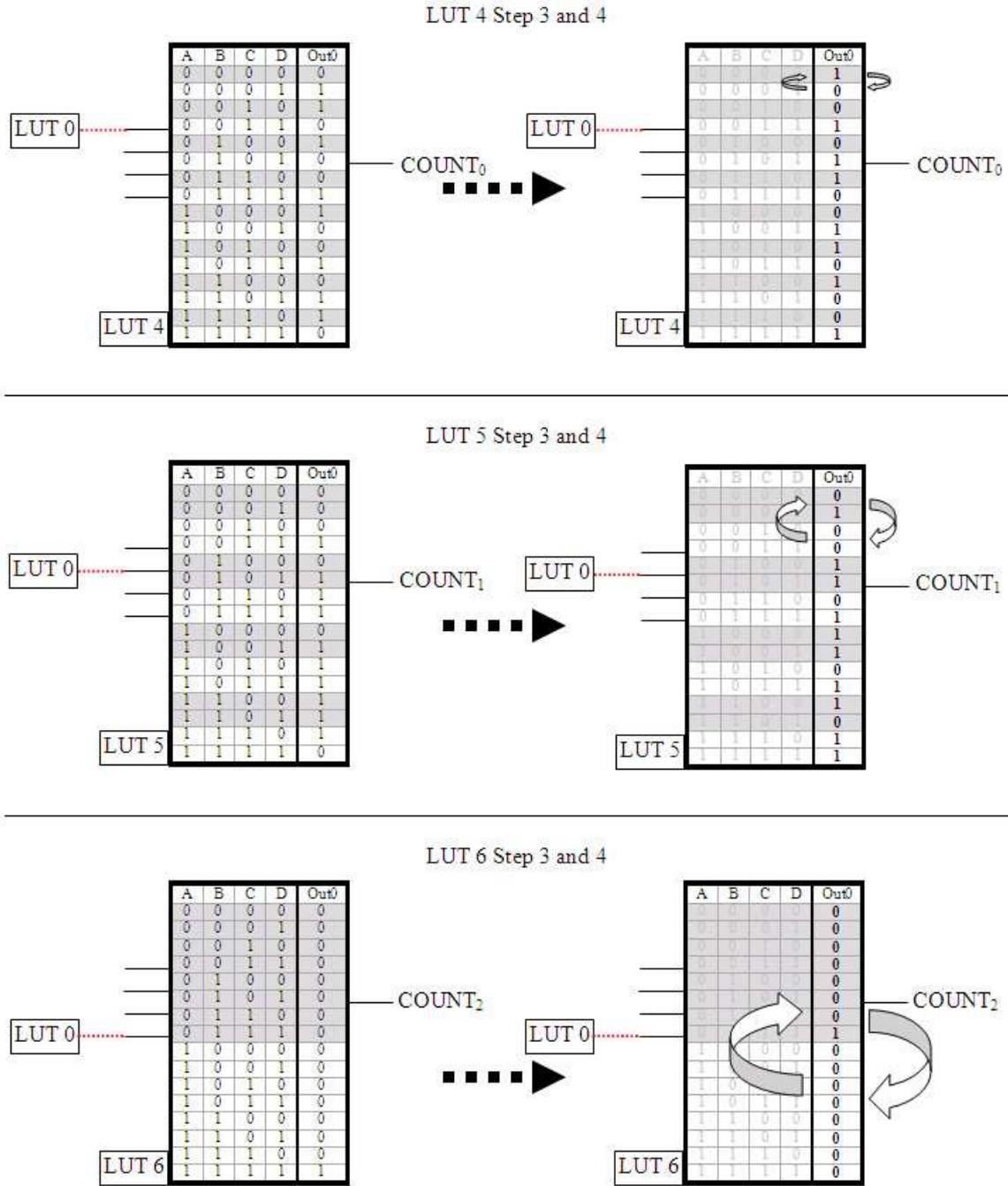


Figure B.3: Demonstration of steps three and four of the LUT inversion algorithm. LUT 0 is connected to LUT 4 on input 0, LUT 5 on input 1 and LUT 6 on input 3. Therefore, neighboring segments of size  $2^0$  (LUT 4),  $2^1$  (LUT 5),  $2^3$  (LUT 6) are swapped. The LUT values on the right represent the final LUT configuration.

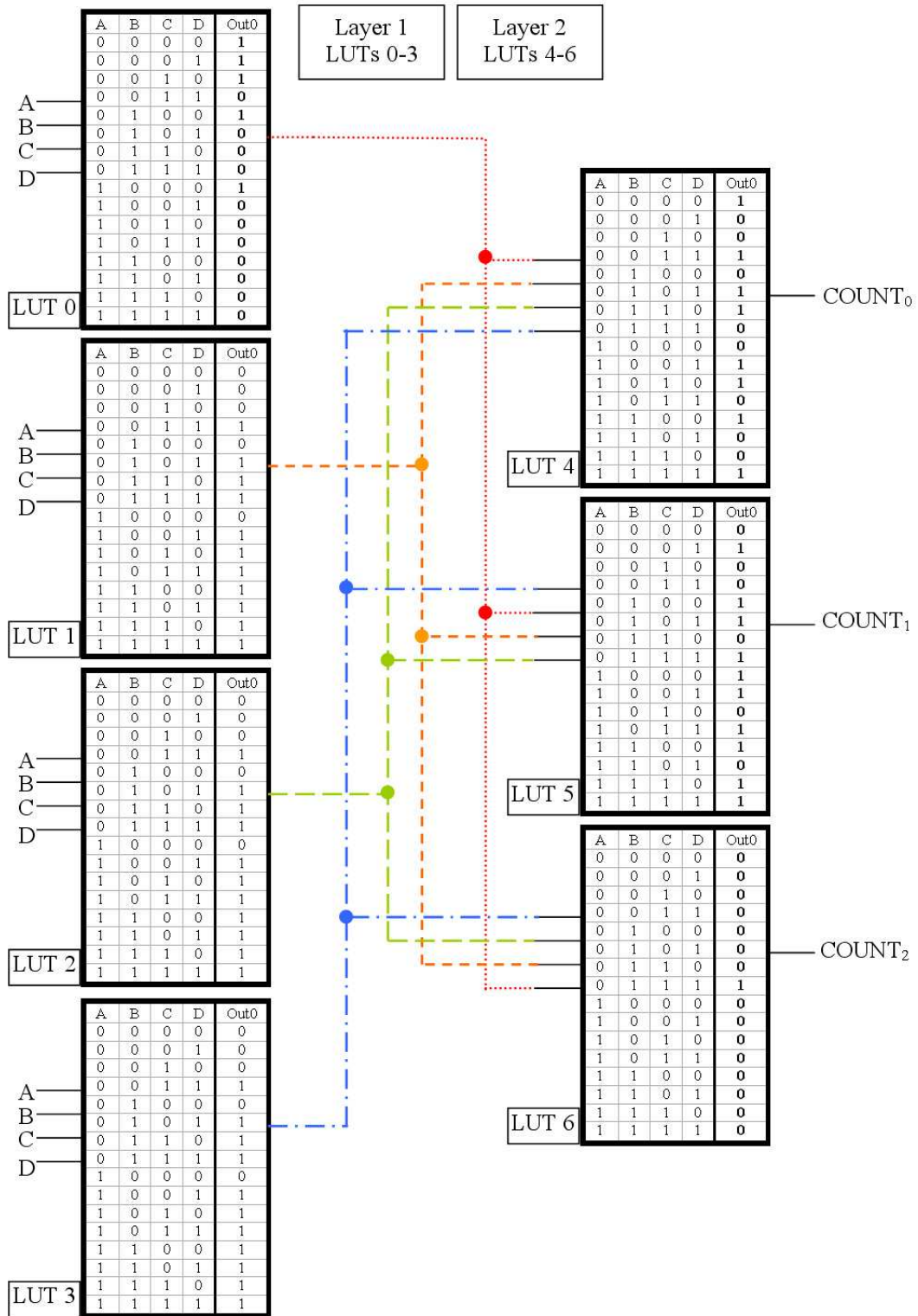


Figure B.4: The final product of performing the LUT inversion algorithm on LUT 0. Every LUT in layer 2, connected to the output of LUT 0, has altered values within the LUT.

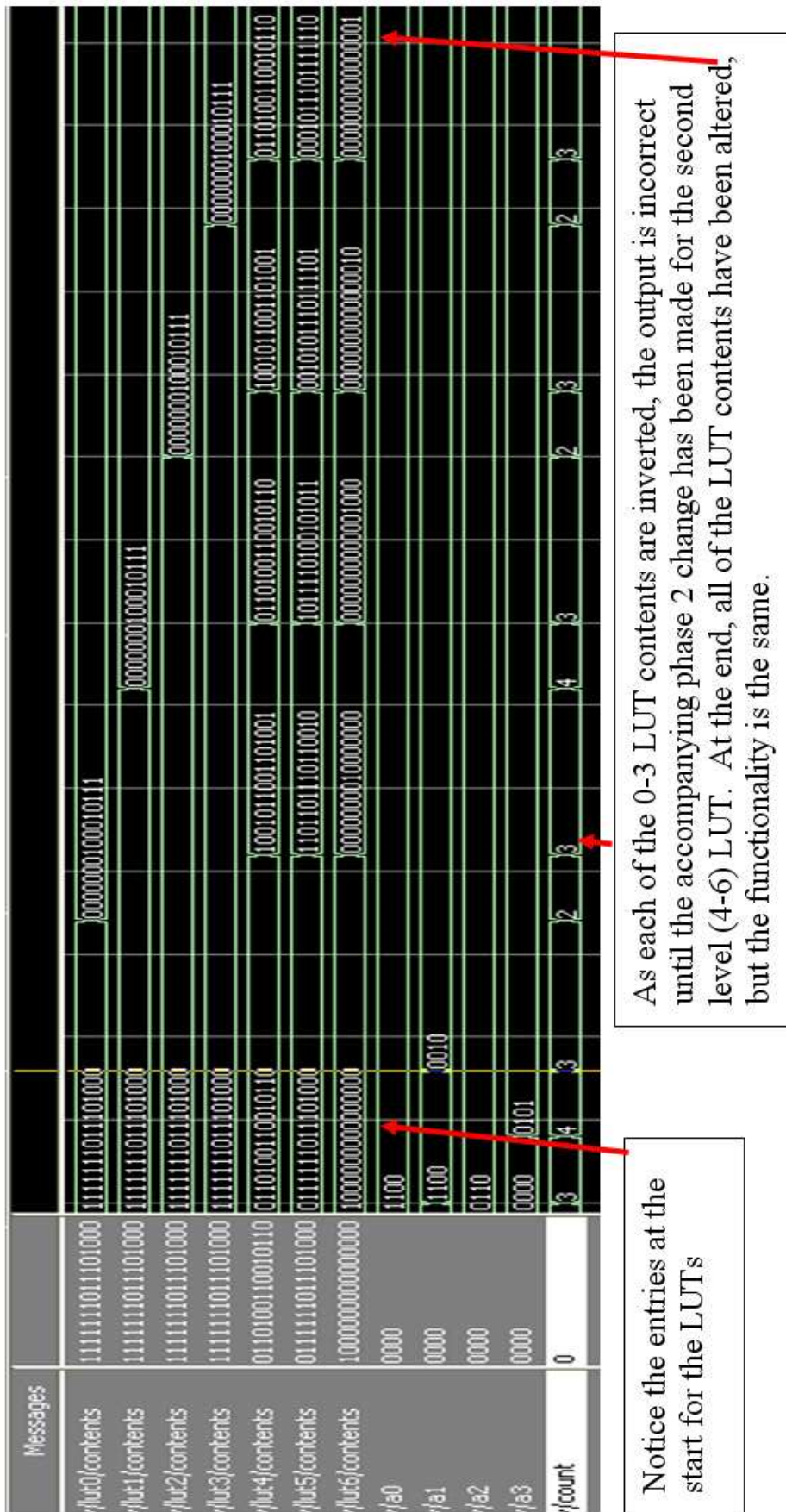


Figure B.5: Test waveforms demonstrating the application of the LUT inversion algorithm on a VHDL LUT circuit model.

### Appendix C. Test Circuit LUT Contents

The following tables specify the LUT contents for implementing the test circuits.

Table C.1: Ripple Carry Adder LUT Values. All slices have the same values for the F and G LUTs. The general makeup is based on the chaining of 8 single bit full adders as outlined in Appendix D.

8 Bit Adder LUT Contents					
Bits 0-7					
Input 0 A	Input 1 B	Input 2 $C_{IN}$	Input 3 Not Used	LUT F Values $C_{OUT}$	LUT G Values SUM
0	0	0	0	0	0
0	0	0	1	0	0
0	0	1	0	0	1
0	0	1	1	0	1
0	1	0	0	0	1
0	1	0	1	0	1
0	1	1	0	1	0
0	1	1	1	1	0
1	0	0	0	0	1
1	0	0	1	0	1
1	0	1	0	1	0
1	0	1	1	1	0
1	1	0	0	1	0
1	1	0	1	1	0
1	1	1	0	1	1
1	1	1	1	1	1

Table C.2: Counter LUT Values. All slices have the same values for the F and G LUTs. The general makeup is based on the chaining of 8 single bit counters with a “carry in” from the previous bits indicating when all previous bits are ‘1’ as outlined in Appendix D. When all previous bits are ‘1’, the current bit flips. For bit 0, the output toggles every clock cycle.

8 Bit Counter LUT Contents					
Bits 0-7					
Input 0 All previous bits = ‘1’	Input 1 Enable Count	Input 2 Q(t)	Input 3 Reset Count	LUT F Values Q(t+1)	LUT G Values All previous and current bits = ‘1’
0	0	0	0	0	0
0	0	0	1	0	0
0	0	1	0	0	1
0	0	1	1	0	0
0	1	0	0	0	0
0	1	0	1	0	0
0	1	1	0	0	1
0	1	1	1	0	0
1	0	0	0	0	0
1	0	0	1	0	0
1	0	1	0	1	1
1	0	1	1	0	0
1	1	0	0	0	1
1	1	0	1	0	0
1	1	1	0	1	0
1	1	1	1	0	0

Table C.3: Comparator LUT Values for stage 1 of the comparator operation. All slices have the same values for the F and G LUTs. There are 4 slices (8 LUTs) in stage 1 of the comparator process. Each Slice compares two concurrent input value pairs,  $A_{x-1} = B_{x-1}$  and  $A_x = B_x$  for  $x = 2^n : n = 0, 1, 2, 3$ .

8 Bit Original Comparator LUT Contents					
Bits 0-7 Stage 1 LUT Contents					
Input 0 $A_x$	Input 1 $B_x$	Input 2 $A_{x+1}$	Input 3 $B_{x+1}$	LUT F Values $A_x = B_x$	LUT G Values $A_{x+1} = B_{x+1}$
0	0	0	0	1	1
0	0	0	1	1	0
0	0	1	0	1	0
0	0	1	1	1	1
0	1	0	0	0	1
0	1	0	1	0	0
0	1	1	0	0	0
0	1	1	1	0	1
1	0	0	0	0	1
1	0	0	1	0	0
1	0	1	0	0	0
1	0	1	1	0	1
1	1	0	0	1	1
1	1	0	1	1	0
1	1	1	0	1	0
1	1	1	1	1	1

Table C.4: Comparator LUT Values for stage 2 of the comparator operation. All slices have the same values for the F-LUTs and G-LUTs. The LUTs are configured in a hierarchical design wherein the previous matches are, themselves matched. In this manner, any pair of original inputs that do not match will cause the final value to be 0. The slice names are listed in the top row to designate which slice handles which level of the hierarchy.

8 Bit Original Comparator LUT Contents						
Bits 0-7 Stage 2 LUT Contents						
Slice	Input 0	Input 1	Input 2	Input 3	LUT F Values	LUT G Values
M0	$A_1 = B_1$	$A_5 = B_5$	$A_0 = B_0$	$A_4 = B_4$	$A_{4-5} = B_{4-5}$	$A_{0-1} = B_{0-1}$
M1	$A_3 = B_3$	$A_7 = B_7$	$A_2 = B_2$	$A_6 = B_6$	$A_{6-7} = B_{6-7}$	$A_{2-3} = B_{2-3}$
L0	$A_{0-1} = B_{0-1}$	$A_{2-3} = B_{2-3}$	$A_{4-5} = B_{4-5}$	$A_{6-7} = B_{6-7}$	$A_{4-7} = B_{4-7}$	$A_{0-3} = B_{0-3}$
L1	$A_{0-3} = B_{0-3}$	$A_{4-7} = B_{4-7}$	'1'	'1'	NA	A=B
	0	0	0	0	1	1
	0	0	0	1	1	0
	0	0	1	0	1	0
	0	0	1	1	1	1
	0	1	0	0	0	1
	0	1	0	1	0	0
	0	1	1	0	0	0
	0	1	1	1	0	1
	1	0	0	0	0	1
	1	0	0	1	0	0
	1	0	1	0	0	0
	1	0	1	1	0	1
	1	1	0	0	1	1
	1	1	0	1	1	0
	1	1	1	0	1	0
	1	1	1	1	1	1

Table C.5: Comparator LUT Values for the functional replacement comparator, phase 1. The overall size of the comparator is reduced to 3 from the 8 used for the original design. All slices have the same values for the F-LUTs and G-LUTs, the only difference being the inputs fed to the LUT. The general layout consists of the first two LUTs, which compare eight pairs of bits from the input and the third LUT, which outputs a 1 if all the pairs match and a 0 otherwise. The LUT relationship is graphically demonstrated in Appendix D.

8 Bit Replacement Comparator LUT Contents					
Bits 0-7 Stage 1 LUT Contents					
LUT	Input 0	Input 1	Input 2	Input 3	LUT Values
G	$A_0$	$B_0$	$A_1$	$B_1$	$A_0 = B_0$ and $A_1 = B_1$
F	$A_2$	$B_2$	$A_3$	$B_3$	$A_2 = B_2$ and $A_3 = B_3$
	0	0	0	0	1
	0	0	0	1	0
	0	0	1	0	0
	0	0	1	1	1
	0	1	0	0	0
	0	1	0	1	0
	0	1	1	0	0
	0	1	1	1	0
	1	0	0	0	0
	1	0	0	1	0
	1	0	1	0	0
	1	0	1	1	0
	1	1	0	0	1
	1	1	0	1	0
	1	1	1	0	0
	1	1	1	1	1

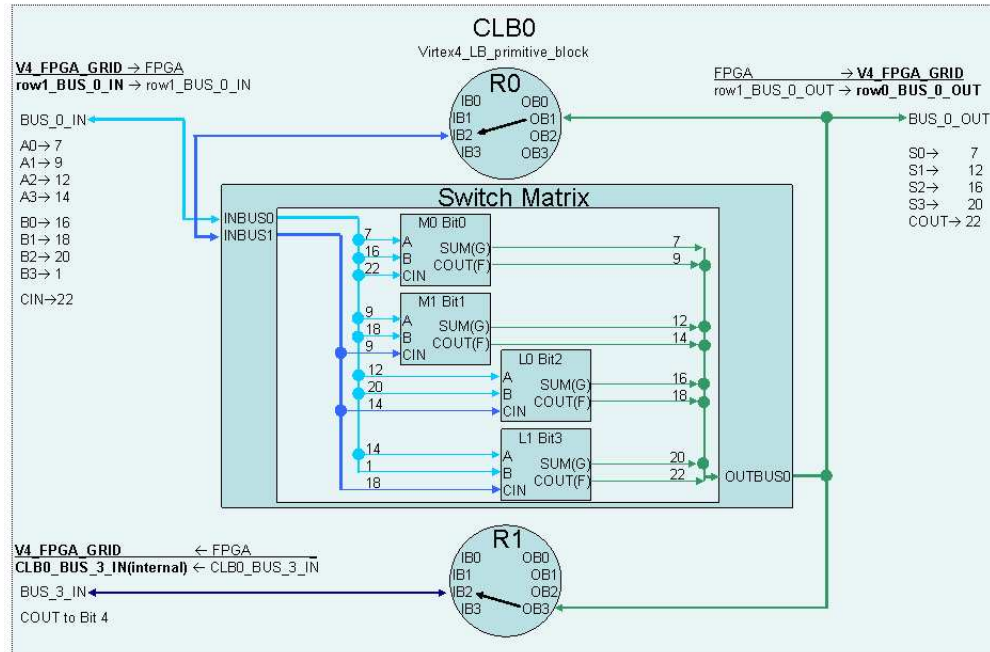
Table C.6: Comparator LUT Values. The Stage 2 is contained completely within the G-LUT of a single slice (the L0 Slice). The LUTs are configured in a hierarchical design wherein the previous matches are, themselves matched. In this manner, any pair of original input matches that are false cause the final value to be 0.

8 Bit Replacement Comparator LUT Contents				
Bits 0-7 Stage 2 LUT Contents				
Input 0 $A_{0-1} = B_{0-1}$	Input 1 $A_{2-3} = B_{2-3}$	Input 2 $A_{4-5} = B_{4-5}$	Input 3 $A_{6-7} = B_{6-7}$	LUT G Values A = B
0	0	0	0	0
0	0	0	1	0
0	0	1	0	0
0	0	1	1	0
0	1	0	0	0
0	1	0	1	0
0	1	1	0	0
0	1	1	1	0
1	0	0	0	0
1	0	0	1	0
1	0	1	0	0
1	0	1	1	0
1	1	0	0	0
1	1	0	1	0
1	1	1	0	0
1	1	1	1	1

## *Appendix D. Test Circuit Schematics*

This appendix documents the test circuit schematics using CLB functional block diagrams. Each circuit is specified as to the LUT and routing usage.

4Bit Adder  
8Bit Adder Bit 0-3



8Bit Adder  
Bit 4-7

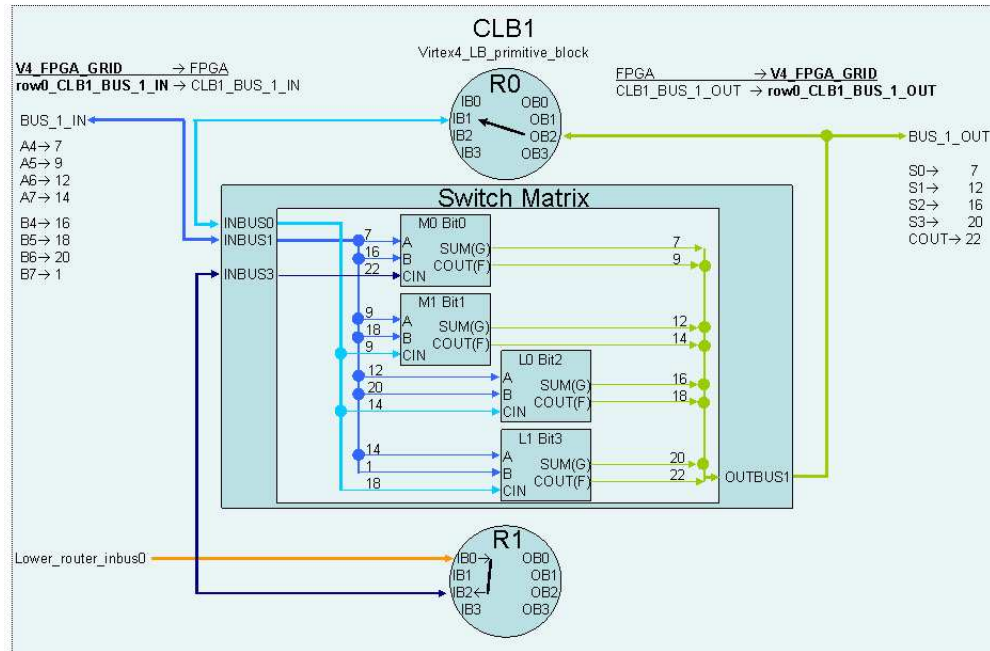
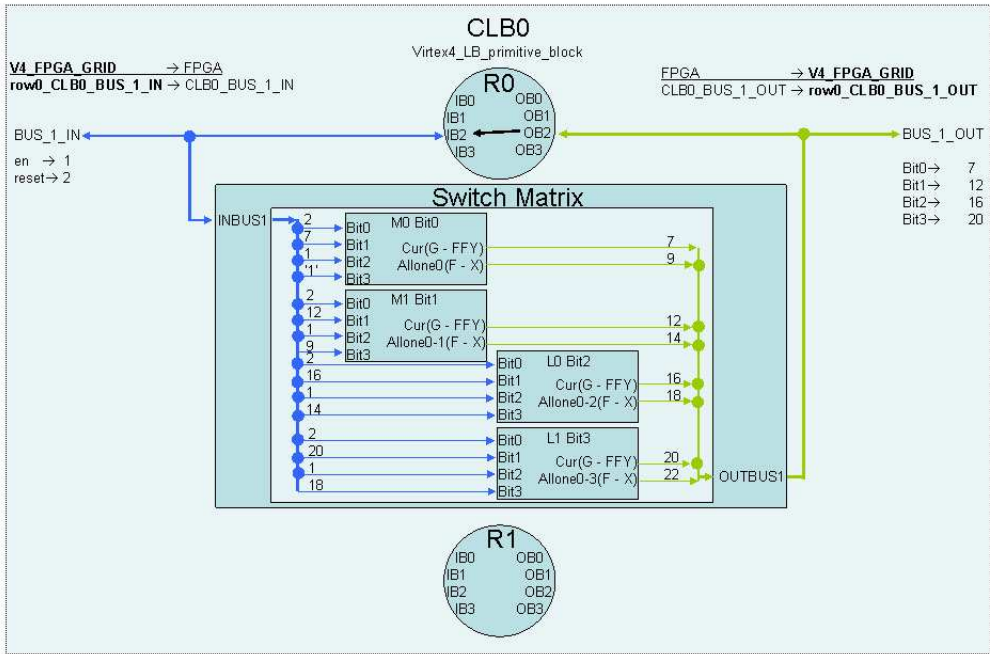


Figure D.1: Test circuit for 8-bit adder, bits 0-7 and the carry in. The 4-bit adder can be implemented using this circuit and taking the outputs for the first four bits of the 8-bit adder and using the single bit carry signal as the final carry out.

4 Bit Counter  
8 Bit Counter (0-3)



#### 8 Bit Counter (4-7)

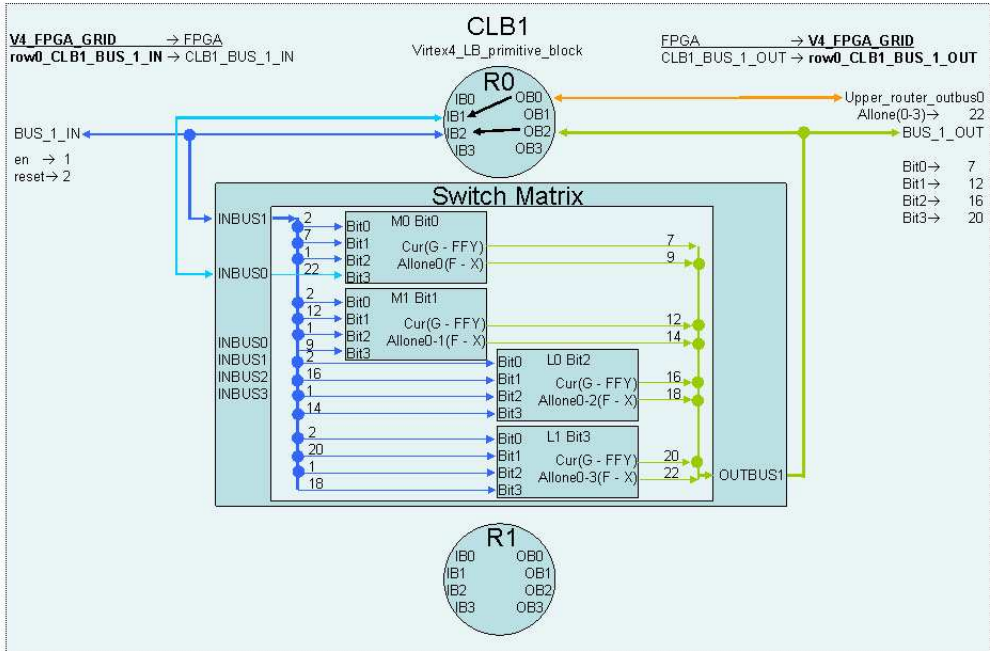
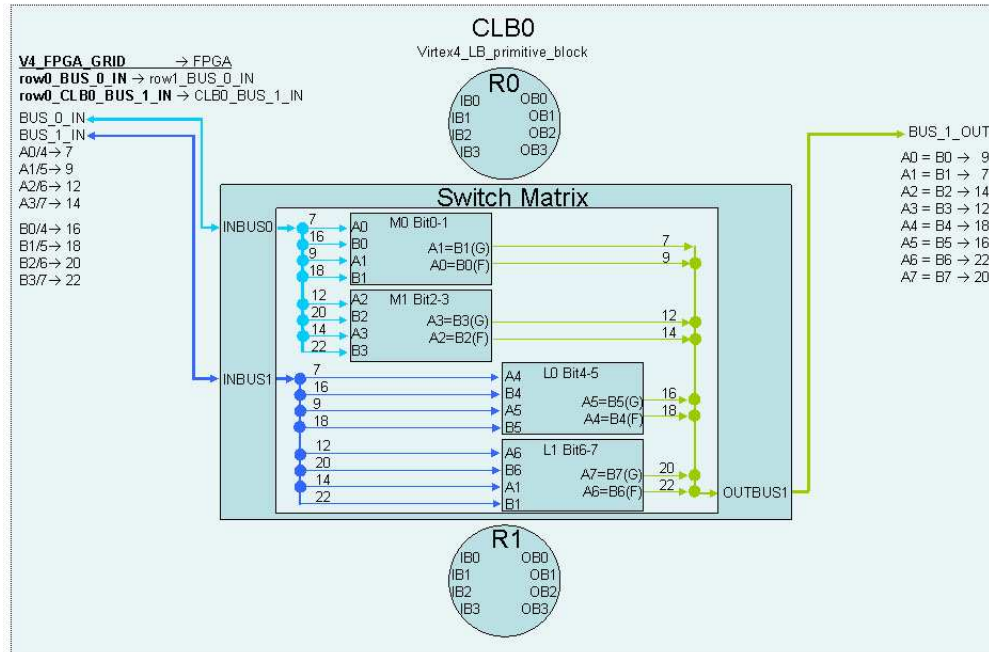


Figure D.2: Test circuit for 8-bit counter. A 4-bit counter can be implemented using this circuit and taking the outputs for the first four bits of the 8-bit adder.

### 8 Bit Comparator Stage 1



### 8 Bit Comparator Stage 2

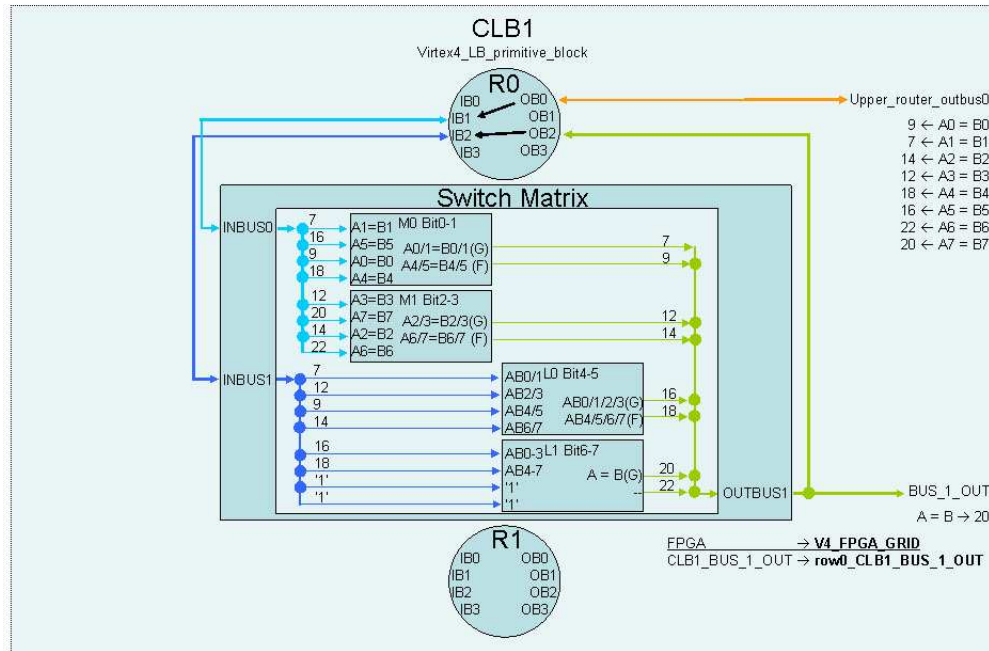


Figure D.3: Test circuit for 8-bit comparator. This is the original circuit before the functional replacement reconfiguration. Figure D.4 demonstrates the post functional replacement circuit.

# 8 Bit Comparator a Stage 1/2

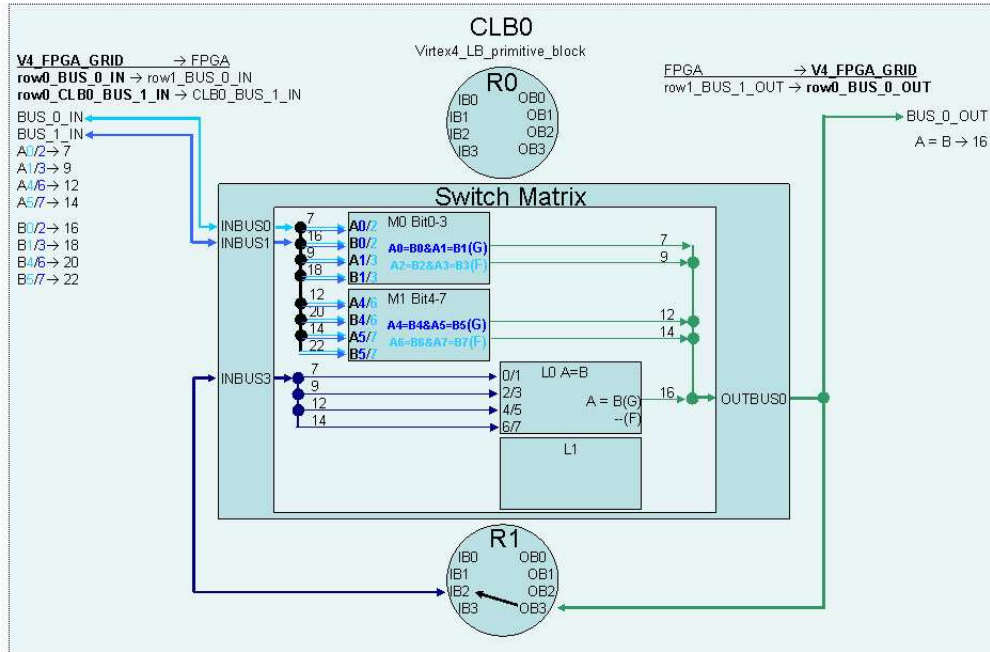


Figure D.4: Test circuit for 8-bit comparator circuit used for functional replacement. This circuit, housed in a single CLB, replaces the original circuit found in Figure D.3. In addition to reducing the total resources needed for circuit implementation, the replacement circuit also spatially obfuscates the output by placing it on a different output bus/pin (bit 16 on Bus 0 instead of bit 20 on Bus 1).

## Appendix E. JTAG Programming Network Example

The following appendix serves to provide a simple example of a JTAG programming network. The example below illustrates the overall arrangement of the boundary scan cells and their relationship to the CLB control registers.

The JTAG Network programs the FPGA CLBs in a serial, sequential manner. Each CLB control register is connected to the other control registers along a 1-bit wide serial data path. For the example demonstrated in Figure E.1, the control registers each hold 626 bits per CLB for the bitstream.

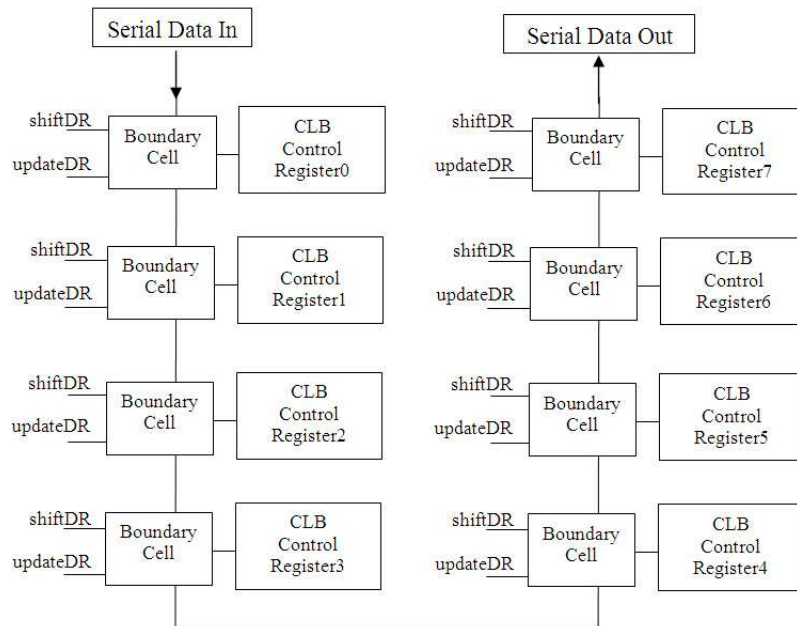


Figure E.1: A simplified, abstract example of a JTAG programming network for an eight CLB FPGA.

The connection to the serial data path is made through a boundary cell, which permits selection of programming or operating modes and contains logic circuits to facilitate the passing of data through the data path to other boundary cells. The JTAG boundary scan cell is described in more detail in Chapter 2. The following steps are followed to accomplish the programming process (some steps/components omitted for clarity).

1. The boundary cells are configured for “pass-through” operation. Chapter 2 contains more data on the specifics of this configuration method.
2. The ShiftDR clock is used to pass the current data stream along the JTAG Network. Once the entire data stream is loaded into the JTAG Network, each boundary cell will contain one bit (which is the first bit to be loaded into the control registers). The process requires one clock cycle per boundary cell in the JTAG Network.
3. Once the boundary cells have their bit of information, the UpdateDR clock is used to latch the data into the most significant bit of the control registers, shifting the data downwards (from most to least significant) similar to a shift register.
4. The previous steps are repeated for each bit within the bitstream. For example, the FPGA developed for this research effort would require the steps to be repeated a total of 626 times.

## Appendix F. CLB Addressable Programming Network Example

The following appendix serves to provide a simple example of the CLB addressable network. The LUT inversion network is nearly identical with the exception of more addressing capabilities through the sub-routers (due to the fact that there are eight times as many LUTs as CLBs) and the substitution of LUT inversion registers instead of the control registers. The example below illustrates the overall arrangement of the chains, routers, and end registers.

The CLB addressable network programs the FPGA CLBs in random access manner. Each CLB control register is connected to a dedicated line from a sub-router module. For the example demonstrated in Figure F.1, the control registers each hold 626 bits per CLB for the bitstream. In the case of the LUT inversion network, the registers would store a total of four bits each for the LUT inversion command.

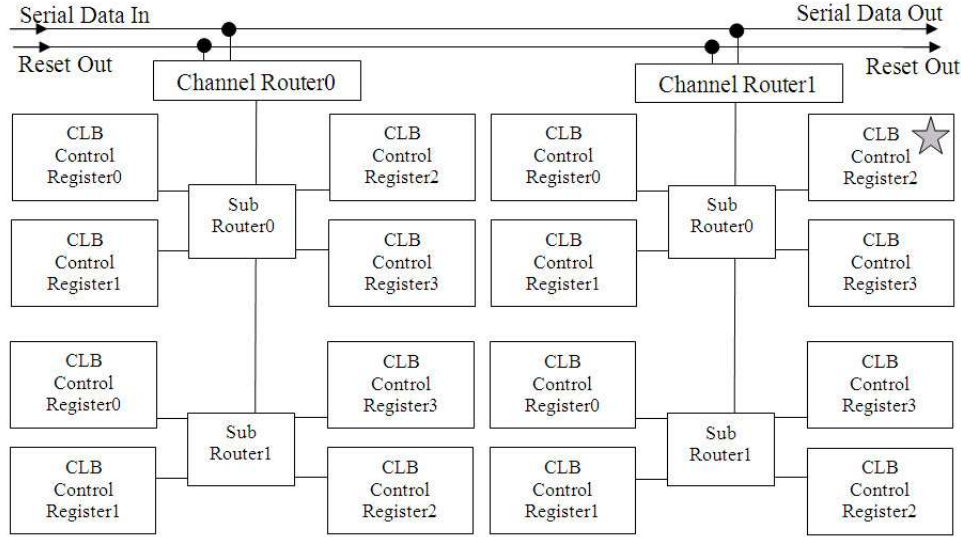


Figure F.1: A simplified, abstract example of a CLB addressable programming network for a 16 CLB FPGA.

Any single CLB Control Register can be targeted for programming by using an addressing system. For example, the CLB Control Register marked with the star corresponds to an address of "1010" (Channel router '1', Sub Router '0', CLB Control Register "10"). Each Channel Router has a "chain" of Sub Routers connected to it which, in turn, have 4 CLBs connected to them. The process of programming a

selected CLB Control Register involves the following steps (some components/steps omitted for clarity).

1. The network is issued a “reset” command.
2. The address is sent along the serial data line. Each Channel Router examines the first bit to determine whether the final target CLB falls within its chain.
3. If the Channel Router’s chain contains the target CLB, then the data stream (minus the Channel Router address portion) is broadcast to the Sub Routers falling within the chain.
4. The Sub Router examines the first bit it receives (but the second bit of the data stream overall) to determine whether the final target CLB is one of the four connected to it.
5. If the Sub Router is connected to the target CLB, then it reads the next two bits from the data stream to determine which CLB will receive the data.
6. Once the target CLB is determined, the data for that CLB is broadcast along the CLB Addressable Network.
7. For example, in order to store the data stream “0101” at the Control Register marked with the star, the following bitstream would be passed through the Serial Data In: “10100101...” where the first four bits represent the CLB Control Register address and the remaining 626 bits represent the data to be stored at the target CLB Control Register.

## Bibliography

1. “Virtex-4 Configuration Guide”, January 2006. URL <http://www.xilinx.com/bvdocs/userguides/ug071.pdf>.
2. “Virtex-4 User Guide”, March 2006. URL <http://www.xilinx.com/bvdocs/userguides/ug070.pdf>.
3. A. Stoica, X. Guo D. Keymeulen M.I. Ferguson, R.S. Zebulum and V. Duong. “Taking evolutionary circuit design from experimentation to implementation: some useful techniques and a silicon demonstration”. *IEEE Proc.-Comput. Digit. Tech.*, 151(4), July 2004.
4. Adrian Stoica, Didier Keymeulen, Ricardo Zebulum and Jason Lohn. “On Polymorphic Circuits and Their Design using Evolutionary Algorithms”. *Applied Informatics*, 351(85), 2002. URL <http://ehw.jpl.nasa.gov/Documents/PDFs/AI2002.pdf>.
5. Alpert, Charles J., Jen-Hsin Huang, and Andrew B. Kahng. “Multilevel Circuit Partitioning”. *Design Automation Conference*, 530–533. 1997. URL [citeseer.ist.psu.edu/article/alpert97multilevel.html](http://citeseer.ist.psu.edu/article/alpert97multilevel.html).
6. Alpert, Charles J. and Andrew B. Kahng. “Geometric Embeddings for Faster and Better Multi-Way Netlist Partitioning”. *Design Automation Conference*, 743–748. 1993.
7. Andrea Lodi, Fabio Campi Andrea Cappelli Roberto Canegallo, Mario Toma and Roberto Guerrieri. “A VLIW Processor With Reconfigurable Instruction Set for Embedded Applications”. *IEEE Journal of Solid-State Circuits*, 38(11):1876–1886, November 2003.
8. Blodget, Brandon, Philip James-Roxby, Eric Keller, Scott Mcmillan, and Prasanna Sundararajan. *A Self-reconfiguring Platform*. 2003. URL <http://www.springerlink.com/content/ack3yyv1xl59pt09>.
9. Brunham, K. and W. Kinsner. “Run-time reconfiguration: towards reducing the density requirements of FPGAs”. *IEEE Canadian Conference on Electrical and Computer Engineering - CCECE01*, 1259–1264. 2001.
10. Burns, J., A. Donlin, J. Hogg, S. Singh, and M. De Wit. “A dynamic reconfiguration run-time system”. *FCCM '97: Proceedings of the 5th IEEE Symposium on FPGA-Based Custom Computing Machines*, 66. IEEE Computer Society, Washington, DC, USA, 1997. ISBN 0-8186-8159-4.
11. Christoff, Joeseph A. *Stabilizing Iraq: DOD Cannon Ensure That U.S.-Funded Equipment Has Reached Iraqi Security Forces*. Report GAO-07-711, United States Government Accountability Office, July 2007.

12. Dueck, S. and W. Kinsner. "Netlist partitioning for FPGA-based run-time reconfiguration". *IEEE Canadian Conference on Electrical and Computer Engineering - CCECE02*, volume 2, 584–590. 2002.
13. G. Bertoni, I. Koren P. Maistri, L. Breveglieri and V. Piuri. "Error Analysis and Detection Procedures for a Hardware Implementation of the Advanced Encryption Standard". *IEEE Transactions on Computers*, 52(4):493–505, April 2003. URL <http://euler.ecs.umass.edu/research/aes03.pdf>.
14. Guang-Ming Wu, Jai-Ming Lin and Yao-Wen Chang. "An Algorithm for Dynamically Reconfigurable FPGA Placement". *IEEE International Conference on Computer Design*. November 2001. URL <http://www.iccd-conference.org/proceedings/2001/12000501.pdf>.
15. H. Bar-El, D. Naccache M. Tunstall, H. Choukri and C. "The Sorcerer's Apprentice Guide to Fault Attacks". *Proceedings of the IEEE*, 94(2):370–382, February 2006.
16. H. Kalte, M. Porrmann and U. Rckert. "System-on-Programmable-Chip Approach: Enabling Online Fine-Grained 1D-Placement". *18th International Parallel and Distributed Processing Symposium 2004 Workshop 3*. February 2004.
17. Hauck, Scott. "Configuration prefetch for single context reconfigurable co-processors". *FPGA '98: Proceedings of the 1998 ACM/SIGDA sixth international symposium on Field programmable gate arrays*, 65–74. ACM Press, New York, NY, USA, 1998. ISBN 0-89791-978-5. URL <http://doi.acm.org/10.1145/275107.275121>.
18. Haug, G. and W. Rosenstiel. "Reconfigurable Hardware as Shared Resource for Parallel Threads". *FCCM '98: Proceedings of the IEEE Symposium on FPGAs for Custom Computing Machines*, 320. IEEE Computer Society, Washington, DC, USA, 1998. ISBN 0-8186-8900-5.
19. Hortal, E. and J. Lockwood. "Automated Method to Generate Bitstream Intellectual Property Cores for Virtex FPGAs". Antwerp, Belgium, August-September 1999. URL <http://www.arl.wustl.edu/~lockwood/publications/PARBIT.FPL.04.pdf>.
20. Jack Jean, Vikram Yavagal Jignesh Shah, Karen Tomko and Robert Cook. "Dynamic Reconfiguration to Support Concurrent Applications". *IEEE Trans. Comput.*, 48(6):591–602, 1999. ISSN 0018-9340.
21. Jiang, Yung-Chuan. "Minimum Communication Cost Approaches for Dynamically Reconfigurable FPGA". *Signals, Circuits and Systems, 2007. ISSCS 2007. International Symposium on*, 1:1–4, 13-14 July 2007.
22. Jones, P., J. Lockwood, and Y. Cho. "A thermal management and profiling method for reconfigurable hardware applications", 2006. URL [citeseer.ist.psu.edu/article/jones06thermal.html](http://citeseer.ist.psu.edu/article/jones06thermal.html).

23. Julian F. Miller, Dominic Job and Vesselin K. Vassilev. "Principles in the Evolutionary Design of Digital Circuits - Part 1". *Journal of Genetic Programming and Evolvable Machines*, 1(1):8–35, 2000. URL <http://www.elec.york.ac.uk/intsys/users/jfm7/jgpem00a.pdf>.
24. K. Kulikowski, A. Taubin, M. Karpovsky. "Robust Codes for Fault Attack Resistant Cryptographic Hardware". September 2005. URL [http://reliable.bu.edu/Projects/FTDC\\_05.pdf](http://reliable.bu.edu/Projects/FTDC_05.pdf).
25. Kamawal, Z. "Experimental Power Analysis Attacks on an FPGA". Oregon State University ECE679 Project, 2004. URL <http://islab.oregonstate.edu/koc/ece679/project/2004/kamawal.pdf>.
26. Kean, T. "Secure Configuration of Field Programmable Gate Array". *IEEE Symposium on Field Programmable Custom Computing Machines (FCCM)*. 2001. URL [http://www.algotronix.com/content/security\\_FPL\\_2001.pdf](http://www.algotronix.com/content/security_FPL_2001.pdf).
27. Koch, Dirk and Jürgen Teich. "Platform-independent methodology for partial re-configuration". *CF '04: Proceedings of the 1st conference on Computing frontiers*, 398–403. ACM, New York, NY, USA, 2004. ISBN 1-58113-741-9.
28. Kwok, Tyrone Tai-On and Yu-Kwong Kwok. "On the Design of a Self-Reconfigurable SoPC Based Cryptographic Engine". *ICDCSW '04: Proceedings of the 24th International Conference on Distributed Computing Systems Workshops - W7: EC (ICDCSW'04)*, 876–881. IEEE Computer Society, Washington, DC, USA, 2004. ISBN 0-7695-2087-1.
29. Lala, P.K. and A. Walker. "An on-line reconfigurable FPGA architecture". *IEEE International Symposium on Defect and Fault Tolerance in VLSI Systems*.
30. Leong, P. H. W., C. W. Sham, H. Y. Wong, W. S. Yuen, M. P. Leong, and W. C. Wong. "A bitstream reconfigurable FPGA implementation of the WSAT algorithm". *IEEE Trans. Very Large Scale Integr. Syst.*, 9(1):197–200, 2001. ISSN 1063-8210.
31. Lucks, S. "Attacking Triple Encryption". *Fast Software Encryption 1998*, 239–253, 1998. URL <http://th.informatik.uni-mannheim.de/People/Lucks/papers/pdf/3des.pdf.gz>.
32. Mak, Wai-Kei and Evangeline F. Y. Young. "Temporal logic replication for dynamically reconfigurable FPGA partitioning". *ISPD '02: Proceedings of the 2002 international symposium on Physical design*, 190–195. ACM, New York, NY, USA, 2002. ISBN 1-58113-460-6.
33. Mermoud, Grégory, Andres Upegui, Carlos Andrés Peña-Reyes, and Eduardo Sanchez. "A Dynamically-Reconfigurable FPGA Platform for Evolving Fuzzy Systems". *8th International Work-Conference on Artificial Neural Networks (Computational Intelligence and Bioinspired Systems)*. Barcelona, Spain, June 2005. URL [http://ic.epfl.ch/webdav/site/ic/shared/article\\_mermoud.pdf](http://ic.epfl.ch/webdav/site/ic/shared/article_mermoud.pdf).

34. Michael L. Bushnell, Vishwani D. Agrawal. *Essentials of Electronic Testing for Digital Memory & Mixed-Signal VLSI Circuits*. Springer, 2000. ISBN 0-7923-7991-8.
35. Multiple. “Federal Information Processing Standards Publication 197”, November 2001. URL <http://csrc.nist.gov/publications/fips/fips197/fips-197.pdf>.
36. Parelkar, M. and K. Gaj. “Implementation of EAX Mode of Operation for FPGA Bitstream Encryption and Authentication”. *Field Programmable Technology, FPT 2005*. Singapore, December 2005. URL <http://mason.gmu.edu/~mparelka/pdfs/fpt05.pdf>.
37. Paul, J., S. Stone, Y. Kim, and R. Bennington. “A Method and Architecture for Real-Time Polymorphic Reconfiguration”. *ICFPT07: International Conference on Field-Programmable Technology 2007*. Kokurakita, Kitakyushu, Japan, December 2007.
38. Qu, Yang, Juha-Pekka Soinen, and Jari Nurmi. “A parallel configuration model for reducing the run-time reconfiguration overhead”. *DATE '06: Proceedings of the conference on Design, automation and test in Europe*, 965–969. European Design and Automation Association, 3001 Leuven, Belgium, Belgium, 2006. ISBN 3-9810801-0-6.
39. Srinivasan, S., A. Gayasen, N. Vijaykrishnan, M. Kandemir, Y. Xie, and M. J. Irwin. “Improving soft-error tolerance of FPGA configuration bits”. *ICCAD '04: Proceedings of the 2004 IEEE/ACM International conference on Computer-aided design*, 107–110. IEEE Computer Society, Washington, DC, USA, 2004. ISBN 0-7803-8702-3.
40. Tiri1, K. and I. Verbauwhede. “Synthesis of Secure FPGA Implementations”. Antwerp, Belgium, June 2004. URL <http://www.ee.ucla.edu/~tiri/files/iwls2004.pdf>.
41. Tomono, Mitsuru, Masaki Nakanishi, Shigeru Yamashita, Kazuo Nakajima, and Katsumasa Watanabe. “An Efficient and Effective Algorithm for Online Task Placement with I/O Communications in Partially Reconfigurable FPGAs”. *IEICE Trans. Fundam. Electron. Commun. Comput. Sci.*, E89-A(12):3416–3426, 2006. ISSN 0916-8508.
42. Tseng, C. “Lock Your Designs with the Virtex-4 Security Solution”. *Xcell Journal Online*, 52, Spring 2005.
43. Webb, J. *Methods for Securing the Integrity of FPGA Configurations*. Master’s thesis, Bradley Department of Electrical and Computer Engineering, Blacksburg, Virginia, August 2006. URL [http://scholar.lib.vt.edu/theses/available/etd-09272006-114810/unrestricted/Webb\\_T](http://scholar.lib.vt.edu/theses/available/etd-09272006-114810/unrestricted/Webb_T)
44. Wiener, Michael J. “Efficient DES Key Search: An Update”, Autumn 1997. URL <http://www.ussrback.com/crypto/cracking-des/cracking-des/chap-11.html>.

45. Wirthlin, Michael J. and Brad L. Hutchings. "Improving functional density using run-time circuit reconfiguration". *IEEE Trans. Very Large Scale Integr. Syst.*, 6(2):247–256, 1998. ISSN 1063-8210.
46. Wollinger, T. and C. Paar. *How Secure Are FPGAs in Cryptographic Applications*. Report 2003/119, German Federal Office for Information Security (BSI)., 2003.
47. Xilinx. *In-Circuit Partial Reconfiguration of RocketIO Attributes*, May 2004. URL <http://www.xilinx.com/bvdocs/appnotes/xapp662.pdf>.
48. Xilinx. "XAPP138 Virtex FPGA Series Configuration and Readback". Application Note, March 2005. URL <http://www.xilinx.com/bvdocs/appnotes/xapp138.pdf>.

## *Vita*

Samuel Stone started his Air Force career as an enlisted member of the 9S100 career field. Enlisting from Spokane, WA in 1997, he followed Basic Training in Lackland by graduating from Technical Training at Goodfellow AFB. While working at the National Air and Space Intelligence Center (NASIC) from 1998 to 2001 Sam started his undergraduate education at Wright State University in Dayton, OH. Sam earned his undergraduate Computer Engineering degree from Wright State University and, through completion of the Airman Education and Commissioning Program (AECPP), was commissioned into the Air Force as a Second Lieutenant in 2003. After serving at Lackland AFB, TX in the 33 IOS Network Defense Flight, Sam applied to and was accepted for the Master's Degree program at the Air Force Institute of Technology (AFIT) in 2006. Following graduation from AFIT, Sam was stationed at Wright Patterson AFB, OH working for the Air Force Research Labs (AFRL).

Permanent address: 2950 Hobson Way  
Air Force Institute of Technology  
Wright-Patterson AFB, OH 45433

REPORT DOCUMENTATION PAGE				Form Approved OMB No. 074-0188	
<p>The public reporting burden for this collection of information is estimated to average 1 hour per response, including the time for reviewing instructions, searching existing data sources, gathering and maintaining the data needed, and completing and reviewing the collection of information. Send comments regarding this burden estimate or any other aspect of the collection of information, including suggestions for reducing this burden to Department of Defense, Washington Headquarters Services, Directorate for Information Operations and Reports (0704-0188), 1215 Jefferson Davis Highway, Suite 1204, Arlington, VA 22202-4302. Respondents should be aware that notwithstanding any other provision of law, no person shall be subject to a penalty for failing to comply with a collection of information if it does not display a currently valid OMB control number.</p> <p><b>PLEASE DO NOT RETURN YOUR FORM TO THE ABOVE ADDRESS.</b></p>					
1. REPORT DATE (DD-MM-YYYY) 27-03-2008		2. REPORT TYPE Masters Thesis		3. DATES COVERED (From – To) September 2006-March 2008	
4. TITLE AND SUBTITLE Anti-Tamper Method for Field Programmable Gate Arrays Through Dynamic Reconfiguration and Decoy Circuits				5a. CONTRACT NUMBER	
				5b. GRANT NUMBER	
				5c. PROGRAM ELEMENT NUMBER	
6. AUTHOR(S)  Stone, Samuel J., Captain, USAF				5d. PROJECT NUMBER JON: 07-152	
				5e. TASK NUMBER	
				5f. WORK UNIT NUMBER	
7. PERFORMING ORGANIZATION NAMES(S) AND ADDRESS(S) Air Force Institute of Technology Graduate School of Engineering and Management (AFIT/EN) 2950 Hobson Way WPAFB OH 45433-7765 DSN: 785-3636				8. PERFORMING ORGANIZATION REPORT NUMBER  AFIT/GE/ENG/08-30	
9. SPONSORING/MONITORING AGENCY NAME(S) AND ADDRESS(ES) Dr. Robert Bennington Research Team Lead AFRL/RYT 2241 Avionics CI WPAFB,OH 45433-7765, AFMC 937--320--9068 ext. 111 Email: robert.bennington@wpafb.af.mil				10. SPONSOR/MONITOR'S ACRONYM(S) AFRL/RYT	
				11. SPONSOR/MONITOR'S REPORT NUMBER(S)	
12. DISTRIBUTION/AVAILABILITY STATEMENT  APPROVED FOR PUBLIC RELEASE; DISTRIBUTION UNLIMITED					
13. SUPPLEMENTARY NOTES					
14. ABSTRACT <p>As Field Programmable Gate Arrays (FPGAs) become more widely used, security concerns have been raised regarding FPGA use for cryptographic, sensitive, or proprietary data. Storing or implementing proprietary code and designs on FPGAs could result in compromise of sensitive information if the FPGA device was physically relinquished or remotely accessible to adversaries seeking to obtain the information. Although multiple defensive measures have been implemented (and overcome), the possibility exists to create a secure design through the implementation of polymorphic Dynamically Reconfigurable FPGA (DRFPGA) circuits. Using polymorphic DRFPGAs removes the static attributes from their design; thus, substantially increasing the difficulty of successful adversarial reverse-engineering attacks. A variety of dynamically reconfigurable methodologies exist for implementations that challenge designers in the reconfigurable technology field.</p> <p>A Hardware Description Language (HDL) DRFPGA model is presented for use in security applications. The Very High Speed Integrated Circuit HDL(VHSIC)language was chosen to take advantage of its capabilities, which are well suited to the current research. Additionally, algorithms that explicitly support granular autonomous reconfiguration have been developed and implemented on the DRFPGA as a means of protecting its designs. Documented testing validated the reconfiguration results, compared original FPGA and DRFPGA, security, power usage, and area estimates.</p>					
15. SUBJECT TERMS Field Programmable Gate Array, dynamic reconfiguration, security, reprogramming, VHDL, active decoy					
16. SECURITY CLASSIFICATION OF:			17. LIMITATION OF ABSTRACT	18. NUMBER OF PAGES	19a. NAME OF RESPONSIBLE PERSON
REPORT	ABSTRACT	c. THIS PAGE			Dr. Yong C. Kim, PhD (ENG)
U	U	U	UU	134	19b. TELEPHONE NUMBER (Include area code) (937) 785-3636; email: yong.kim@afit.edu